Eiffel library to generate Java bytecodes

Master Thesis

Author(s):
Gisel, Daniel

Publication date:
2003

Permanent link:
https://doi.org/10.3929/ethz-a-005115106

Eiffel Library to generate Java Bytecodes

—

Diploma Thesis

Daniel Gisel

Supervised by Prof. Dr. Bertrand Meyer and Karine Arnout

ETH Zürich

May 26 – September 25, 2003

Abstract

The goal of this diploma thesis was to create an Eiffel library to generate Java bytecodes. The library is divided into two abstraction levels. The lower level is a direct mapping from the structure of the Java class file to an object structure. The higher level contains generators to simplify the creation of classes, methods, fields, constants and attributes. There is also a block structure to compose the bytecode instructions. To test the library and give an example how it can be used, I wrote a simple Java compiler that uses the library as back-end.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of my diploma project was to create an Eiffel library to generate Java bytecodes, namely an API that can be used to build an object structure representing the content of a Java class file, dump it to the disk and execute it on a JVM.

The library is structured into two different abstraction levels. The lower level is a direct mapping of the structures from the Java class file to a simple object structure. Every part in the class file has its corresponding object. With this kind of structure the user has the possibility to control every single byte of the class file. But he also has to specify everything because there is hardly any automation, except the index computation.

The higher level defines some structure generators on the one hand and a block structure to compose the bytecode instructions on the other hand. The structure generators simplify the creation of classes, methods, fields, attributes and constants by collecting the needed information and then generate complete structures. The block structure supports several code constructs like loops, conditionals or the try-catch construct and generates the corresponding code. The higher abstraction level always generates structures of the lower level.

The library can be used in a back-end of a compiler that targets Java bytecodes. As source language one can imagine every programming language.

To give an example of how the library can be used in a compiler back-end and also to test the library and its usability I wrote a simple Java compiler that can compile a subset of the Java programming language to Java bytecodes.

The remainder of this chapter explains the terminology I use and gives a general introduction into the Java Programming Language Environment. Then, I discuss the design (Chapter 2) and the implementation (Chapter 3) of the library. In Chapter 4, I explain the compiler and the used subset of the Java programming language. In the last two chapters I discuss (Chapter 5) the design, implementation and usability of the library and of the compiler and give an outlook (Chapter 6) about which parts of the library should be adapted, where the usability can be improved and how it should be used. The appendix contains an API of the library (Appendix A) and the grammar of the Java programming language (Appendix B).

1.1 Terminology

This report does not introduce any new terminology. Nevertheless it is not simple to understand everything in the correct way because I worked with two programming lan-

guages, Eiffel and Java, that sometimes have different names for similar constructs or the same name for different things.

I tried to use the correct terminology for both languages. This means that whenever I am talking about Java and the class files, I use the Java terminology and whenever I talk about the design and the implementation of the library I use the Eiffel terminology. Of course the classes and features in the library that have a close relation to the class file have the same name like the corresponding construct in the JVM definition [LY99].

1.2 Java Programming Language Environment

Even if I implement my whole project in the Eiffel programming language, the target of my library is the Java class file that can be executed on the Java Virtual Machine (JVM). This is the reason why I write here an introduction about the Java Programming Language Environment [GM96].

1.2.1 Beginnings of Java

The Java programming language originated as part of a research project to develop advanced software for a wide variety of network devices and embedded systems. The goal was to develop a small, reliable, portable, distributed, real-time operating platform. When the project started, C++ was the language of choice. But over time the difficulties encountered with C++ grew to the point where the problems could best be addressed by creating an entirely new language platform. Design and architecture decisions drew from a variety of languages such as Eiffel, Smalltalk, Objective C, and Cedar/ Mesa. The result is a language platform that has proven practical for developing secure, distributed, network-based end-user applications in environments ranging from network-embedded devices to the World-Wide Web and the desktop.

1.2.2 Java technology

If someone talks about Java [Jav], he probably talks about the Java programming language. But Java is not only a high-level programming language, the name Java stands also for a whole platform. Unlike many other programming languages that are either compiled or interpreted, the Java programming language is both compiled and interpreted.

First, the compiler translates a program from the Java programming language into a platform-independent intermediate language called Java bytecodes. The interpreter (JVM) parses and runs each Java bytecode instruction on the computer. The program is compiled only once, but interpreted each time it is executed.

The Java bytecodes help make “write once, run anywhere” possible. As long as a computer has an implementation of a JVM, a program in Java bytecodes can run on it, independently of the operating system or the hardware of the computer and independently of the system the program has been written and compiled.

1.2.3 The Java class file format

The Java bytecodes are stored in Java class files. A class <class_name> is written to the class file with the name <class_name>.class. A class file consists of a stream of 8-

bits bytes. Bigger quantities (16-bits, 32-bits, 64-bits) are built by two, four, or eight consecutive 8-bits bytes. Multibyte data items are always stored in big-endian order, where the high bytes come first.

The complete definition of the class file format is written in the Java Virtual Machine Specification [LY99].

Chapter 2

Design of the library

The goal of my diploma project was to build an Eiffel library to generate Java class files that can be executed on a JVM. On one hand, the library should be an abstraction that makes the code generation for the user as simple as possible, but on the other hand it should also be possible for the user to manipulate every single byte in the class file to guarantee as much flexibility as possible. To fulfill both requirements, the library is built upon two abstraction levels.

The first, lower abstraction level (Section 2.1) is nearly a one-to-one mapping of the structures in a Java class file to an object-oriented class hierarchy. Nearly every item in the file has its corresponding object that defines its properties and also has the ability to write the item to the class file. References between items are represented as references between the objects and are resolved to indices by writing the items to the class file.

The second, higher abstraction level (Section 2.2) is built upon the classes of the lower level. It consists of generators that simplify the creation and the composition of the several parts of the class file. There are generators to build classes, constant pools, methods, fields and all kind of attributes. For the simple generation of code into the class file, there is a block structure that can be composed recursively and that generates complete code attributes.

The API of the library can be found in Appendix A.

2.1 Lower abstraction level

The design of the lower abstraction level is very close to the structures of a Java class file and allows the user to access and manipulate nearly every single byte. The built class hierarchy is quite similar to the structure of the class file (Figure 2.1). Most classes represent a certain part in the class files, but there are also a few classes with a more general purpose, described here.

There is a class `CLASS_FILE_COMPONENT` that is a proper ancestor for all classes in the library that represent a part of the class file. `CLASS_FILE_COMPONENT` defines the deferred feature `dump_component (file: JAVA_CLASS_FILE)` and the corresponding feature `is_dump_allowed (file: JAVA_CLASS_FILE): BOOLEAN` that is used in the precondition of `dump_component`. By implementing / redefining these two features, the descendants make sure that they may write their own content to the class file (`dump_component`), but only if they have a state that allows a dump (`is_dump_allowed`).

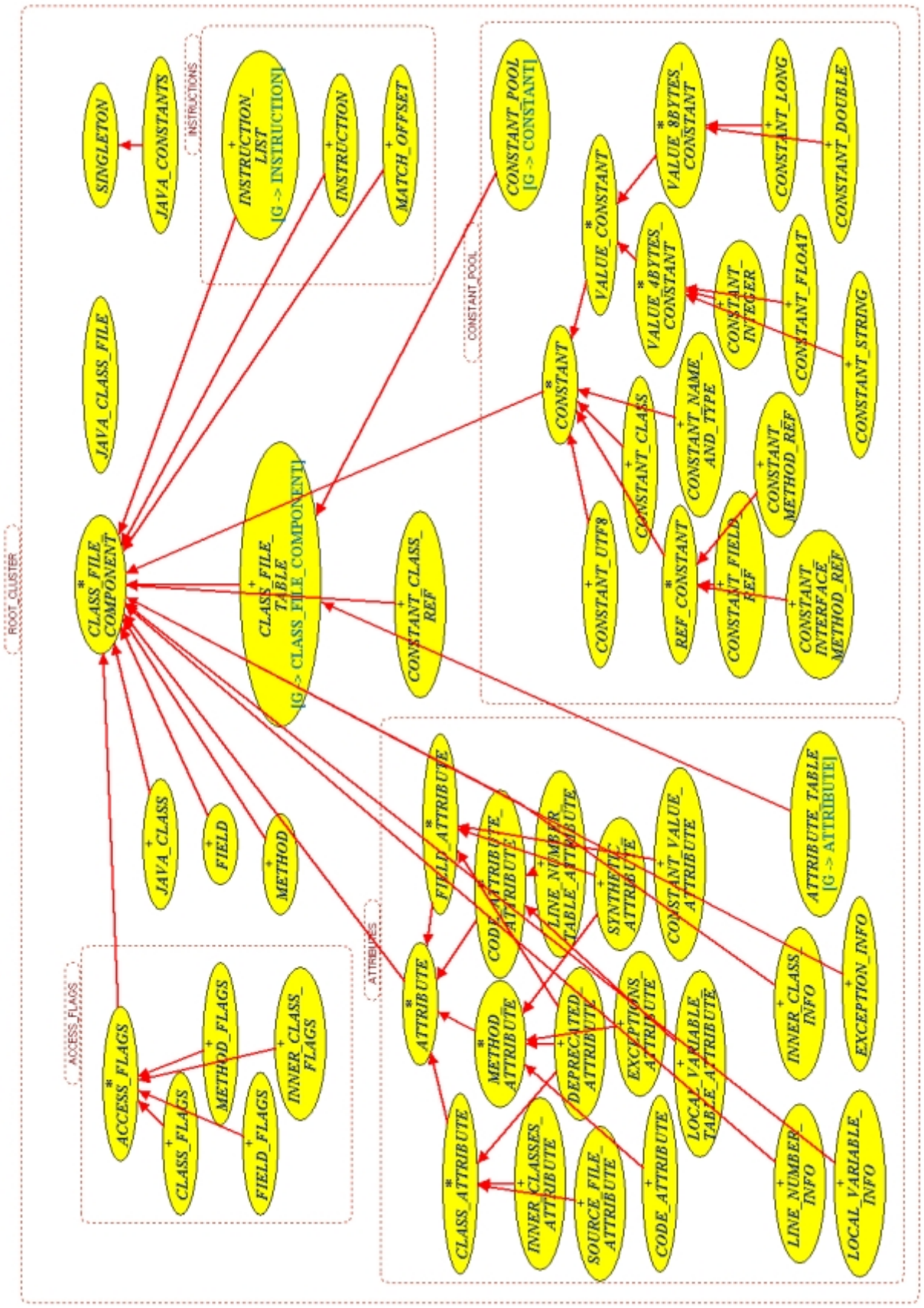

Figure 2.1: Class hierarchy.

In a Java class file one can find a lot of tables, consisting of zero or more variable-sized items. All these tables have the same structure: first the number of items is written to the class file, followed by items, written sequentially, without padding or alignment. In the library, the class CLASS_FILE_TABLE is used to create these tables. CLASS_FILE_TABLE is an ARRAYED_LIST of class file components that also inherits from the class CLASS_FILE_COMPONENT so that it can write its own content to the class file.

The class JAVA_CLASS_FILE is the interface between the class hierarchy representing a Java class file and the real file that is written to the disk. It inherits from RAW_FILE but defines new features to write in big-endian order to the file. There are features to write unsigned and signed bytes and shorts, ints, longs, floats, doubles and also strings in the UTF-8 format.

The class CONSTANT_CLASS_REF represents a reference to a class constant in the constant pool (Section 2.1.2). It is used in tables of references to class constants like in the interface table of the class (Section 2.1.1).

The JAVA_CONSTANTS class is defined to store constants used in the library. It implements together with the class SINGLETON a singleton pattern [AB, GHJV94] so that only one instance of this class can be created.

2.1.1 Class

The main class in the library is JAVA_CLASS. It represents a whole Java class or in other words, it is the counterpart of the content of a whole class file. Of course it also inherits from CLASS_FILE_COMPONENT and a call to feature dump_component in JAVA_CLASS causes the library to write the whole class file to the disk.

The content of the JAVA_CLASS is quite similar to the ClassFile structure (Figure 2.2) defined in the JVM specification [LY99]. It defines attributes with corresponding set procedures for all the items in the ClassFile structure except the magic item that always has the value 0xCAFEBABE.

The minor and the major versions are represented as INTEGER values.

The constant pool is modeled by a reference to the class CONSTANT_POOL [CONSTANT] (Section 2.1.2). This class counts also the items of the constant pool; it exposed the feature constant_pool_count.

The access flags are defined by the class CLASS_FLAGS (Section 2.1.6).

this_class and super_class are represented by references to the corresponding class constants in the constant pool. The indices that are written to the class file are computed directly by the constant pool.

The interfaces and the interfaces_count are represented by a CLASS_FILE_TABLE with the actual generic parameter CONSTANT_CLASS_REF that represents a reference to a constant pool entry.

The field and the method table and their count are also defined as a CLASS_FILE_TABLE with actual generic parameter FIELD (Section 2.1.3) and METHOD (Section 2.1.4), respectively.

The class attributes are stored in an ATTRIBUTE_TABLE with the actual generic parameter CLASS_ATTRIBUTE (Section 2.1.5).

```
ClassFile {
  u4 magic;
  u2 minor_version;
  u2 major_version;
  u2 constant_pool_count;
  cp_info constant_pool[constant_pool_count-1];
  u2 access_flags;
  u2 this_class;
  u2 super_class;
  u2 interfaces_count;
  u2 interfaces[interfaces_count];
  u2 fields_count;
  field_info fields[fields_count];
  u2 methods_count;
  method_info methods[methods_count];
  u2 attributes_count;
  attributes_info attributes[attributes_count];
}
```

Figure 2.2: ClassFile structure.

2.1.2 Constant pool

The constant pool is a table of structures representing various string constants, class and interface names, field names, and other constants that are referred to within the ClassFile structure and its substructures. The format of each constant pool table entry is indicated by a tag (Table 2.1).

For every constant type in the constant pool, there is a corresponding class in the library. All these classes inherit (directly or indirectly) from the class `CONSTANT` that defines the basic content of the constants. It has an attribute `tag` to specify the type of the constant, an attribute `index` that is set to the index of the constant in the constant pool and an attribute `size` that returns the logical size of the constant. Almost all constants occupy one index item in the constant pool and so their size is 1. But the constants for the 8 byte values (`CONSTANT_Long` and `CONSTANT_Double`) occupy two index items (size = 2) which means that if a `CONSTANT_Long` or a `CONSTANT_Double` has index i, the next constant in the pool has index i + 2 instead of i + 1.

To manage this strange design[1] of the indices, the class `CONSTANT_POOL` is defined. It inherits from `CLASS_FILE_TABLE` and has an additional attribute `constant_pool_count` that returns the size of the constant pool ($\sum$ CONSTANT.size + 1). The class `CONSTANT_POOL` also redefines the features `force` and `extend` to correctly recompute the size of the constant pool by inserting new constants and to compute the correct indices of the constants. All other features of the feature clauses 'Element change', 'Removal' and

[1] Even in the JVM specification [LY99] is written: 'In retrospect, making 8-byte constants take two constant pool entries was a poor choice.'

Constant Type	*Tag Value*	*Size*
CONSTANT_Class	7	1
CONSTANT_Fieldref	9	1
CONSTANT_Methodref	10	1
CONSTANT_InterfaceMethodref	11	1
CONSTANT_String	8	1
CONSTANT_Integer	3	1
CONSTANT_Float	4	1
CONSTANT_Long	5	2
CONSTANT_Double	6	2
CONSTANT_NameAndType	12	1
CONSTANT_Utf8	1	1

Table 2.1: Constant Types.

'Transformation' are exported to `NONE` to avoid wrong indices and constant pool sizes.

The different types of constants redefine the attributes `tag` and `size` of `CONSTANT` to their corresponding constant values (Table 2.1). The rest is defined in the following manner:

CONSTANT_CLASS: It contains an attribute `name_index` (with the corresponding set procedure) that holds a reference to a `CONSTANT_UTF8` with the class name. If the `CONSTANT_CLASS` represents an array[2], the referenced `CONSTANT_UTF8` contains the descriptor of the array type.

CONSTANT_NAME_AND_TYPE: It defines the signature of a field or a method and contains the two attributes `name_index` and `descriptor_index` (with the corresponding set procedures) that both hold a reference to a `CONSTANT_UTF8` with, respectively, the name and the descriptor.

CONSTANT_UTF8: This class is used to represent constant string values in the class file. It contains the three queries `length`, `value` and `value_unicode`. The attributes `value` and `value_unicode` and their set procedures are used to read and store the string of this constant (`value` for ASCII strings and `value_unicode` for Unicode strings). `length` returns the length (in bytes) of the UTF-8 representation of the string.

REF_CONSTANT: This deferred class represents a reference to a field, a method or an interface method. It has the attributes `name_and_type_index` and `class_index`. The `name_and_type_index` is a reference to a `CONSTANT_NAME_AND_TYPE` that defines the signature of the field, the method or the interface method. The `class_index` is a reference to a `CONSTANT_CLASS` representing the class or the interface that contains the field, the method or the interface method.

CONSTANT_FIELD_REF: This structure represents a reference to a field. It inherits from `REF_CONSTANT` that defines the content of `CONSTANT_FIELD_REF`.

[2]Arrays are objects in the Java environment.

The `CONSTANT_CLASS` connected to the `class_index` may be either a class type or an interface type.

CONSTANT_METHOD_REF: This structure represents a reference to a method. It inherits from `REF_CONSTANT` that defines the implementation of `CONSTANT_METHOD_REF`. The `CONSTANT_CLASS` connected to the `class_index` must be a class type, not an interface type.

CONSTANT_INTERFACE_METHOD_REF: This structure represents a reference to an interface method. It inherits from `REF_CONSTANT` that defines the content of `CONSTANT_INTERFACE_METHOD_REF`. The `CONSTANT_CLASS` connected to the `class_index` must be an interface type, not a class type.

VALUE_CONSTANT: This deferred class does not have any content. It is only created for a structural purpose of the class hierarchy. It is a proper ancestor of all constant pool classes that may store constant values of the Java data types (int, long, float, double, String).

VALUE_4BYTES_CONSTANT: This deferred class represents the constants that are accessed by a 4-byte value (int, float, String). It has mainly a structural purpose.

CONSTANT_INTEGER: It is used to represent `int` constants in the class file. The attribute `value` and its corresponding set procedure allow, respectively, to read and to set the value of this constant.

CONSTANT_FLOAT: It is used to represent `float` constants in the class file. The attribute `value` and its corresponding set procedure allow to read and to set the value of this constant.

CONSTANT_STRING: This constant is used to represent constant objects of type `String`. It defines the attribute `string_index` (with the corresponding set procedure) which is a reference to a `CONSTANT_UTF8` containing the value of the string.

VALUE_8BYTES_CONSTANT: This deferred class represents the 8-byte numeric constants (long and double). It mainly has a structural purpose.

CONSTANT_LONG: This class represents a `long` constant in the class file. The attribute `value` and its corresponding set procedure allow to read and to set the value of this constant.

CONSTANT_DOUBLE: This class represents a `double` constant in the class file. The attribute `value` and its corresponding set procedure allow to read and to set the value of this constant.

2.1.3 Fields

The fields of a Java class are stored in a table of the class file, represented in the library by a `CLASS_FILE_TABLE[FIELD]` which is stored in the attribute `fields` of the `JAVA_CLASS` (Section 2.1.1). The structure of the items in the field table are defined in the JVM specification [LY99] by the `field_info` structure (Figure 2.3).

```
field_info {
   u2 access_flags;
   u2 name_index;
   u2 descriptor_index;
   u2 attributes_count;
   attribute_info attributes[attributes_count];
}
```

Figure 2.3: field_info structure.

The class `FIELD` inherits from `CLASS_FILE_COMPONENT` to be able to write its content to the class file (`dump_component`, `is_dump_allowed`). Its design is quite similar to the `field_info` structure.

The attribute `access_flags` and its corresponding set procedure handle a reference to a `FIELD_FLAGS` object (Section 2.1.6) that defines the field access right.

The `name_index` and the `descriptor_index` (with their set procedures) are references into the constant pool (`CONSTANT_UTF8`, Section 2.1.2). They contain the name respectively the descriptor of the field.

The attribute with the name `attributes` holds a reference to an `ATTRIBUTE_TABLE` with the actual generic parameter `FIELD_ATTRIBUTE` (Section 2.1.5). These attributes can give additional information about the field.

2.1.4 Methods

Like the fields, the methods of a class are stored in a table in the class file. In my library this table is referenced in the `methods` attribute of the `JAVA_CLASS` (Section 2.1.1) and has the type `CLASS_FILE_TABLE [METHOD]`.

```
method_info {
   u2 access_flags;
   u2 name_index;
   u2 descriptor_index;
   u2 attributes_count;
   attribute_info attributes[attributes_count];
}
```

Figure 2.4: method_info structure.

The entries of that class file table, the `method_info` items (Figure 2.4), are mapped to the class `METHOD` in the library. This class inherits from `CLASS_FILE_COMPONENT` to be

able to dump its content to the class file. It also defines several attributes to implement the items of the `method_info` structure.

The attribute `access_flags` references a `METHOD_FLAGS` object (Section 2.1.6) that defines the access rights of the method.

The attributes `name_index` and `descriptor_index` hold references to constant pool constants (`CONSTANT_UTF8`, Section 2.1.2) that contain the name and descriptor of the method.

To set and access the attributes table of the method, one has to use the features `attributes` and `set_attributes` of type `ATTRIBUTES_TABLE [METHOD_ATTRIBUTE]` (Section 2.1.5). The attribute table of the method is very important because the program code of the method is stored into the `CODE_ATTRIBUTE`.

2.1.5 Attributes

Attributes in the class file are used to give additional information[3]. They all have the same general format (Figure 2.5) that is mapped to the deferred class `ATTRIBUTE`. There is a query `name_index` with the type `CONSTANT_UTF8` which is a reference to a constant pool UTF-8 constant that contains the attribute name. The query `length` returns the length (in bytes) of the subsequent information (`info[]`) of the attribute.

```
attribute_info {
   u2 attribute_name_index;
   u4 attribute_length;
   u1 info[attribute_length];
}
```

Figure 2.5: attribute_info structure.

As part of the class file specification there are certain predefined attributes (see [LY99]). But it is also possible to write a compiler that emit class files containing new attributes in the attribute tables of the class file structures. My library only contains the predefined attributes but if someone wants to create a new attribute, he only has to implement a class that inherits from `ATTRIBUTE`.

The attributes can be attached to the `ClassFile` (Section 2.1.1), to the `field_info` (Section 2.1.3), to the `method_info` (Section 2.1.4) and to the `Code_attribute` (Section 2.1.5) structures, but not all of the predefined attributes may be attached to all structures. To make it impossible to add an attribute to a structure that does not support it, there are four deferred classes inheriting from `ATTRIBUTE` in the library (`CLASS_ATTRIBUTE`, `FIELD_ATTRIBUTE`, `METHOD_ATTRIBUTE` and `CODE_ATTRIBUTE_ATTRIBUTE`). These classes do not have any content, they are only used for a structural purpose. The attributes classes inherit from one or more of these classes depending on the structures they may be attached to.

[3] An attribute in the Java class file is not the same as an attribute in Eiffel.

For every predefined attribute there is a corresponding class in the library. Some attribute classes also need a helper class (with the suffix `_INFO`) to realize a table using the `CLASS_FILE_TABLE`.

Constant value attribute

The constant value attribute is a fixed-length attribute and may be attached to the `field_info` structure (Section 2.1.3), where the corresponding field must be static (the `ACC_STATIC` bit must be set (Section 2.1.6)), otherwise the constant value attribute will be silently ignored by the JVM.

In the library, the constant value attribute is mapped to the class `CONSTANT_VALUE_ATTRIBUTE`. The attribute length is set to 2 and the attribute `constantvalue_index` holds a reference to a constant pool entry (`VALUE_CONSTANT`, Section 2.1.2) that contains the value of the constant field.

Code attribute

The code attribute is a variable-length attribute used in the attribute table of the `method_info` structure (Section 2.1.4). It contains the JVM instructions and auxiliary information for the method. If a method is neither native nor abstract, it must have exactly one code attribute in its attribute table.

```
Code_attribute {
   u2 attribute_name_index;
   u4 attribute_length;
   u2 max_stack;
   u2 max_locals;
   u4 code_length;
   u1 code[code_length];
   u2 exception_table_length;
   {    u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
   }    exception_table[exception_table_length];
   u2 attributes_count;
   attribute_info attributes[attributes_count];
}
```

Figure 2.6: Code_attribute structure.

In the library, the code attribute is represented by the class `CODE_ATTRIBUTE` that is quite similar to the `Code_attribute` structure (Figure 2.6). There is also a class `EXCEPTION_INFO` to present the content of the exception table of a code attribute.

The items max_stack, max_locals, code_length and the code array are defined in the class INSTRUCTION_LIST [INSTRUCTION] (Section 2.1.7); they can be set and accessed with the features set_code and code.

The exception_table is mapped to a CLASS_FILE_TABLE with the actual generic parameter EXCEPTION_INFO. This class has features to get and set the four items of the exception table. The attributes start_pc, end_pc and handler_pc hold references to INSTRUCTION objects where the instruction of end_pc is the last instruction in the range (the index of the first instruction after the range is computed by dumping the content to the class file). The attribute catch_type references a constant pool entry representing a class (CONSTANT_CLASS, Section 2.1.2).

The attributes array of the code attribute is implemented by an ATTRIBUTE_TABLE [CODE_ATTRIBUTE_ATTRIBUTE] and the features attributes and set_attributes.

Exceptions attribute

The exceptions attribute is a variable-length attribute, attached to a method_info (Section 2.1.4), that indicates which checked exceptions a method may throw.

The exception attribute is represented in the library by the class EXCEPTION_ATTRIBUTE. Apart from the attribute_name_index and the attribute_length it only contains the exception_index_table that is mapped to a CLASS_FILE_TABLE with the actual generic parameter CONSTANT_CLASS_REF to reference the class entries in the constant pool (Section 2.1.2) of the potential exceptions.

Inner classes attribute

The inner classes attribute is attached to the ClassFile structure (Section 2.1.1) and has a variable length. It indicates that the constant pool refers to a class or an interface that is not member of a package.

```
InnerClasses_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 number_of_classes;
  {   u2 inner_class_info_index;
      u2 outer_class_info_index;
      u2 inner_name_index;
      u2 inner_class_access_flags;
  } classes[number_of_classes];
}
```

Figure 2.7: InnerClasses_attribute structure.

The inner classes attribute is represented by the class INNER_CLASSES_ATTRIBUTE in the library. Its content, the classes table, can be accessed with the features classes and set_classes of type CLASS_FILE_TABLE [INNER_CLASS_INFO].

The class INNER_CLASS_INFO manages the content of the table items. The attributes inner_class_info_index and outer_class_info_index are references to constant pool entries of the type CONSTANT_CLASS (Section 2.1.2). The inner_name_index is also a reference to the CONSTANT_UTF8 constant pool entry that contains the name of the inner class if it has any. The inner_class_access_flags of the type INNER_CLASS_FLAGS (Section 2.1.6) define the access permissions of the inner class.

Synthetic attribute

The synthetic attribute is a fixed length attribute in the attributes table of the ClassFile (Section 2.1.1), the field_info (Section 2.1.3) and the method_info (Section 2.1.4) structure. It is used to indicate that a class member does not appear in the source code.

The class SYNTHETIC_ATTRIBUTE does not have any content apart from the inherited features from the class ATTRIBUTE.

Source file attribute

The source file attribute has a fixed length and is attached to the ClassFile (Section 2.1.1) structure. It is used to indicate the name of the source file from which the class was compiled.

In the library, the class SOURCE_FILE_ATTRIBUTE has an attribute source_file_index that holds a reference to a CONSTANT_UTF8 object in the constant pool (Section 2.1.2); it contains the name of the source file.

Line number table attribute

The line number table attribute is a variable-length attribute in the attribute table of the code attribute (Section 2.1.5). It may be used by debuggers to determine which part of the Java virtual machine code array corresponds to a given line number in the original source file.

The class LINE_NUMBER_TABLE_ATTRIBUTE contains a table (line_number_table) of type CLASS_FILE_TABLE [LINE_NUMBER_INFO]. The class LINE_NUMBER_INFO has the two attributes (with corresponding set procedures) start_pc that is a reference to an instruction (Section 2.1.7) and line_number that holds the corresponding line number in the source file.

Local variable table attribute

The local variable table attribute is a variable-length attribute of a code attribute (Section 2.1.5). It may be used by debuggers to determine the value of a given local variable during the execution of a method. It specifies the place where a local variable is stored, depending on the program counter.

The class LOCAL_VARIABLE_TABLE_ATTRIBUTE contains a CLASS_FILE_TABLE [LOCAL_VARIABLE_INFO] that stores all the information about the local variables. The class LOCAL_VARIABLE_INFO has the attributes start_pc and end_pc with the type INSTRUCTION (Section 2.1.7) to define a range in the code. The attributes name_index and descriptor_index are references to CONSTANT_UTF8 objects in the constant pool

(Section 2.1.2); they contain the name and descriptor of the local variable. The attribute `index` holds the index of the local variable in the local variable array.

Deprecated attribute

The deprecated attribute is a fixed-length attribute in the attributes table of the `ClassFile` (Section 2.1.1), the `field_info` (Section 2.1.3) and the `method_info` (Section 2.1.4) structures. A class, interface, method, or a field may be marked using a deprecated attribute to indicate that the class, interface, method, or field has been superseded.

The class `DEPRECATED_ATTRIBUTE` does not have any feature apart from the features inherited from the class `ATTRIBUTE`.

2.1.6 Access flags

The access flags in a class file are used in the `ClassFile` (Section 2.1.1), the `field_info` (Section 2.1.3), the `method_info` (Section 2.1.4) and the `InnerClasses_attribute` (Section 2.1.5) structures. In the library they are represented by the deferred class `ACCESS_FLAGS`.

The attribute `access_flags` from the class `ACCESS_FLAGS` stores the value of the flags. With the features `set_access_flags` and `add_access_flags` one can set respectively change the access flags. The feature `is_flag_set` allows a bit-wise checking of the access flags.

Value	*Name*	*Class*	*Field*	*Method*	*InnerClass*
0x0001	ACC_PUBLIC	X	X	X	X
0x0002	ACC_PRIVATE		X	X	X
0x0004	ACC_PROTECTED		X	X	X
0x0008	ACC_STATIC		X	X	X
0x0010	ACC_FINAL	X	X	X	X
0x0020	ACC_SUPER	X			
0x0020	ACC_SYNCHRONIZED			X	
0x0040	ACC_VOLATILE		X		
0x0080	ACC_TRANSIENT		X		
0x0100	ACC_NATIVE			X	
0x0200	ACC_INTERFACE	X			X
0x0400	ACC_ABSTRACT	X		X	X
0x0800	ACC_STRICT			X	

Table 2.2: Access flags.

To allow only legal flags (Table 2.2), the library provides the classes `CLASS_FLAGS`, `FIELD_FLAGS`, `METHOD_FLAGS` and `INNER_CLASS_FLAGS`. All these classes inherit from `ACCESS_FLAGS` but redefine the feature `valid_flags` to allow only the flags they support. The values of the different flags and features handling the flags validity are defined in the class `JAVA_CONSTANTS`.

2.1.7 Instructions

The Java virtual machine instructions stored in the code attribute (Section 2.1.5) of the methods are represented in the library by the class `INSTRUCTION`. Every instance of this class represents one single instruction. The opcode of the instruction must be set at creation time of the object and cannot be changed later on.

The class `INSTRUCTION` provides several features to set all possible arguments of an instruction. But the preconditions of these features restrict their use to meaningful cases only (i.e. if the instruction has an opcode that supports this argument). The feature `length` returns the length of the instruction in bytes.

The class `MATCH_OFFSET` is a helper class for the class `INSTRUCTION` and is used to represent the instruction `lookupswitch`. It holds the value that has to be matched and a reference to the target instruction if the value matches.

The class `INSTRUCTION_LIST` is a descendant of `LINKED_LIST` and `CLASS_FILE_COMPONENT`. It is used to compose the instructions and compute their addresses. It also stores the code length in bytes, the `max_stack` and the `max_locals` values. The features of the feature clauses 'Element change', 'Removal' and 'Transformation' are either exported to `NONE` or redefined to add the needed functionality to compute the addresses.

2.2 Higher abstraction level

The goal of the higher abstraction level is to simplify the generation of Java class files for the user. By using the lower abstraction level (Section 2.1) the user has to create and add every detail to a class file. Also the management of the constant pool is quite sophisticated if one wants to avoid multiple UTF-8 constants containing the same string.

The higher abstraction level builds structures of the lower level, but its generators automatize the process as much as possible. For example by specifying the name of a method, the method generator creates the needed UTF-8 constant and adds it to the constant pool, using the constant pool generator.

The classes of the higher abstraction level can be divided into two parts. On one hand there are the structure generators (Section 2.2.1) that support the user by creating constant pools, methods, fields, attributes or even whole class files. On the other hand there are the code generators (Section 2.2.2) that help the user by composing the code of the methods.

2.2.1 Structure generators

The structure generators are used to create the structures of the Java class file. They are defined in the cluster `generators`. All generators need access to the constant pool of the class because they have to create and add constants for their components to reference to. But the generators do not work directly with the constant pool, they all use the constant pool generator that is also described in this section.

Class generator

The class generator is represented in the library by the class `CLASS_GENERATOR`. It creates a `JAVA_CLASS` object that can be accessed with the attribute `java_class`. To be

able to work on this object, the class generator holds references to a constant pool generator (`constant_pool_gen`) and an attribute generator (`attribute_gen`). The class can be manipulated by setting the version numbers (`set_major_version`, `set_minor_version`), the name of the super class (`set_super_class`) and the value of the access flags (`set_access_flags`). It is also possible to add fields (`add_field`), methods (`add_method`), attributes (`add_attribute`) and names of interfaces that the class implements (`add_interface`). The command `add_empty_constructor` creates an empty constructor method with the given access flags and adds it to the created class.

To set the super class and to add interfaces, one only has to pass a string with the name; the class generator creates the corresponding constants in the constant pool and sets the references. For setting the access flags one only has to specify the access flags value and the class generator handles the access flags object.

Constant pool generator

The class `CONSTANT_POOL_GENERATOR` can be used to create and manage the constant pool accessible through the attribute `constant_pool`.

For every kind of constants of the constant pool, there are two visible features in the constant pool generator. The status report feature with the name `has_constant_<constant name>` indicates if there is already a constant in the constant pool that contains the passed values. The access feature `constant_<constant name>` returns a reference to a constant in the constant pool that contains the passed values. If there was already such a constant in the pool before the feature was invoked, a reference to this constant is returned. Else, a new constant is created and added to the pool and its reference is returned.

Using this constant pool generator guarantees that there are no multiple similar constants in the pool, but a constant can be referenced several times.

Field generator

The class `FIELD_GENERATOR` is used to create a field of a Java class. The created field can be accessed by the attribute `field`. The 'element change' features allow the following specifications of the field. `set_name` and `set_descriptor` set the name respectively the descriptor of the field. They also insert the corresponding UTF-8 constants to the constant pool. `set_access_flags` defines the access flags by updating the access flags object of the field. With the method `add_attribute` one can add attributes to the field.

Method generator

The method generator, represented by the class `METHOD_GENERATOR` that generates the `method` is quite similar to the class `FIELD_GENERATOR`. It also has procedures `set_name`, `set_descriptor`, `set_access_flags` and `add_attribute`. There is one additional procedure `add_code` that adds a code attribute to the method; it makes sure there are enough local variables reserved to store all parameters passed to the method. The procedure `add_exception` allows to specify the names of the exceptions the method may throw and produces the corresponding constants in the constant pool.

Attribute generator

The attribute generator (class `ATTRIBUTE_GENERATOR`) is used to create empty attributes that already have a reference to their name index constant in the constant pool. There is for every attribute defined in the JVM specification [LY99] a function that creates the corresponding attribute objects and adds its name index reference.

2.2.2 Code generators

The code generator classes of the higher abstraction level simplify the creation and composition of the code attributes in the methods of a class file. They build blocks for commonly used structures like loops, if-then-else constructs, the try-catch or the switch construct. Of course there is also a block for linear code. The code generators are defined in the sub-cluster `code_blocks` of the cluster `generators`.

Block

The basic unit of the code generator is a block represented by the deferred class `BLOCK`. The block specifies an interface that must be implemented by all its descendants, representing the different code structures, so that they can be composed recursively.

The most important feature of the block is the query `code_attribute` that generates the whole code attribute for the block including the computation of the maximal stack size and the maximal number of local variables.

The query `is_complete` indicates if the block has enough information to be accessed.

All the information of the code attribute can also be accessed separately. `exception_table`, `line_number_table` and `local_variable_table` return the contents of the line number table of the code attribute and the contents of the line number and local variable table attributes that can be attached to the code attribute. The features `add_line_number_info` and `add_local_variable_info` can be used to add line number and local variable info entries to the current block.

The query `instructions` returns a `LINKED_LIST` of `INSTRUCTION` that contains all instructions of the block. But there is no address computation done. `first_instruction` and `last_instruction` return the first, respectively the last instruction of the block that can be used as target for jump or branch instructions.

`max_locals` computes the maximal number of used local variables for the operations in the block. But this number is only computed from the code. If the method header has unused arguments there have to be more local variables reserved than the code really needs.

For the maximal stack depth computation there are several features. The maximal stack depth of a block can be accessed by the feature `max_stack`. The query `end_stack` computes the stack depth at the end of the block in relation to the stack depth at the beginning of the block. If the user does not want the block to computes its maximal stack depth and the stack depth at the end of the block it is possible to set these values by using the procedures `set_max_stack` and `set_end_stack` that also set the attributes `is_max_stack_user_set` respectively `is_end_stack_user_set` to true. To reset these attributes to false, the procedures `compute_max_stack` and `compute_end_stack` can be used.

The query `is_return_block` returns true if the code of the block returns to the caller of the method in all possible cases.

Code block

The code block (Figure 2.8) is a very simple block. It represents a linear section of JVM code. This means that it must not contain any jump or branch instruction because then the computation of the maximal stack depth would become impossible without control flow analysis.

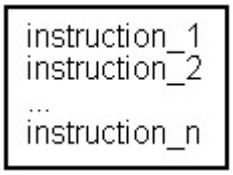

Figure 2.8: Code block.

The class `CODE_BLOCK` inherits form `BLOCK` and from `LINKED_LIST [INSTRUCTION]`. To add and manipulate the instructions in the code block one has to use the features of the class `LINKED_LIST`.

Concat block

The concat block (Figure 2.9) is also quite simple. It is used to concatenate two blocks.

It is represented by the class `CONCAT_BLOCK` that inherits from `BLOCK`. The features `set_first_block` and `set_second_block` are used to set the attributes `first_block` and `second_block` that store the content of the concat block.

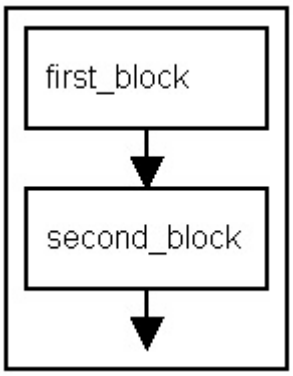

Figure 2.9: Concat block.

While block

The while block (Figure 2.10) is used to realize a while statement of the Java programming language in bytecodes. It is implemented in the class `WHILE_BLOCK` that inherits from `BLOCK`.

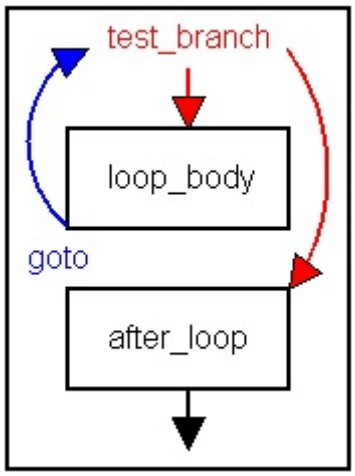

Figure 2.10: While block.

The attributes `loop_body` and `after_loop` store the blocks containing the code that is executed in, respectively after the loop. The after, loop block has to be set to make it possible for the branch instruction to compute its jump target.

The choice of the branch instruction can be made with the feature `set_test_branch` that takes as argument the opcode of the branch instruction. The values that the branch instruction tests must be loaded on the stack before the while block and in the loop body.

Do-while block

The do-while block (Figure 2.11), implemented by the class `DO_WHILE_BLOCK`, allows a simple mapping from the do-while statement in the Java programming language to Java bytecodes.

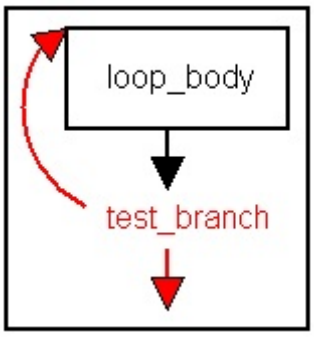

Figure 2.11: Do-while block.

It contains a block that can be accessed by the attribute `loop_body` that is executed in the loop. At the end of the loop body, there must be instructions to load the values onto the stack that should be tested in the branch operation (`test_branch`). The kind of branch operation is defined by the procedure `set_test_branch` that takes the opcode as argument.

In comparison to the while block (Section 2.2.2), the do-while block does not need an after loop block, because there is no jump to the code after the loop.

If-then block

The if-then block (Figure 2.12) builds the block structure for an if-then statement that can be dumped to Java bytecodes.

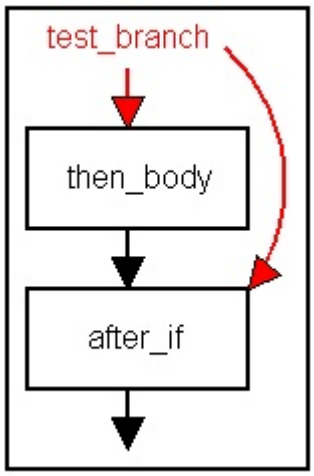

Figure 2.12: If-then block.

It contains two blocks that are stored in the attributes `then_body` and `after_if` and can be set with the procedures `set_then_body` and `set_after_if`. The then body is executed if the condition of the if statement is true and after if is used for the branch instruction to compute its target for the jump. The branch instruction is stored in the attribute `test_branch` and its opcode can be set with the procedure `set_test_branch`. The values the branch instruction compares have to be loaded on the stack before the if-then block is executed.

If-then-else block

The if-then-else block (Figure 2.13) is used to realize an if-then-else statement in the Java bytecodes.

The block stored in `then_body` is executed if the condition has result true. If the result is false, the block stored in `else_body` is executed. The block stored in `after_if` is used to enable the goto instruction, which is appended to the then body to compute its jump target.

The branch operation can be accessed through the attribute `test_branch` and its opcode can be specified with the procedure `set_opcode`.

If `then_body` contains a block that returns in every case to the caller (indicated by the query `is_return_block`) then the after if block does not have to be specified because no goto is needed after the then body.

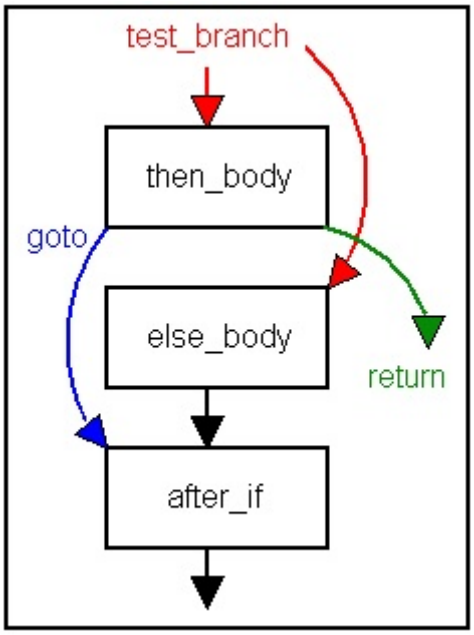

Figure 2.13: If-then-else block.

Switch block

The switch block (Figure 2.14) can be used to map the switch statement of the Java programming language to Java bytecodes.

The user can choose between `tableswitch` and `lookupswitch`, the two different JVM instructions for this statement, by specifying the corresponding opcode with the procedure `set_switch_instruction`.

The procedure `set_after_switch` sets the after switch block stored in `after_switch`. With the procedure `add_case_blocks` it is possible to add cases to the switch statement. To define a case the following information is needed: the value for the label of the case, indication if it is the default block, the code block with the code of the case and the specification if the case block contains a break statement. If the case block is the default block, its value is irrelevant. If the code block is void, the indication of wether it contains a break statement becomes irrelevant.

If all case blocks are return blocks and a default block exists, then the after switch block does not have to be specified because there is no jump that needs it as target.

Try-catch block

The try-catch block (Figure 2.15) realizes the try-catch statement from the Java programming language that is used to catch and handle exceptions.

It contains three different kinds of blocks: The first is the `try_block` that contains the code that can throw an exception. Then there are the exception handlers that can be specified with the procedure `add_exception_handler`. The arguments of this procedure are first the type of the exception that has to be caught; second the block with the code that has to be executed if this exception occurs. The third kind is the block stored in `after_try_catch` that is executed after the whole try-catch statement. It is used for the

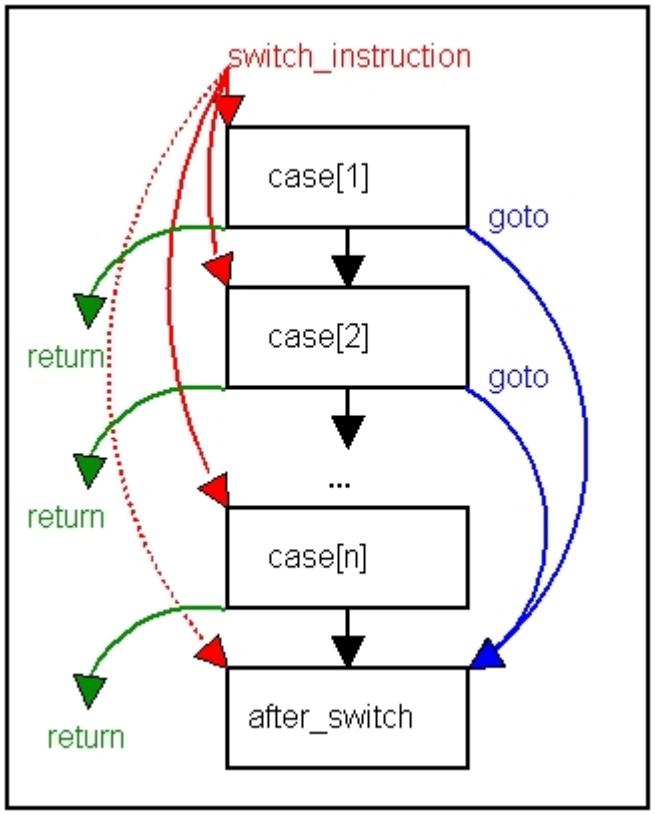

Figure 2.14: Switch block.

computation of the jump targets of the goto instructions that are appended to the try body and the catch bodies. If the try block and all catch blocks are return blocks then the after try block does not have to be specified.

Print block

The print block can be used to write something to the standard output using the Java methods `println` or `print` of the class `PrintStream`[4].

Before executing this block, the data for the standard output has to be loaded on the top of the stack. One also has to specify the type of the date with one of the following procedures: `write_string`, `write_object`, `write_int`, `write_float`, `write_double`, `write_long`, `write_char`, `write_boolean` or `write_char_array`. Further, one must specify if a line break should be appended (`print` or `println`).

If one only wants to write a line break, no data has to be put on the stack and the procedure `write_new_line` should be called.

[4]In Java: System.out.println() or System.out.print()

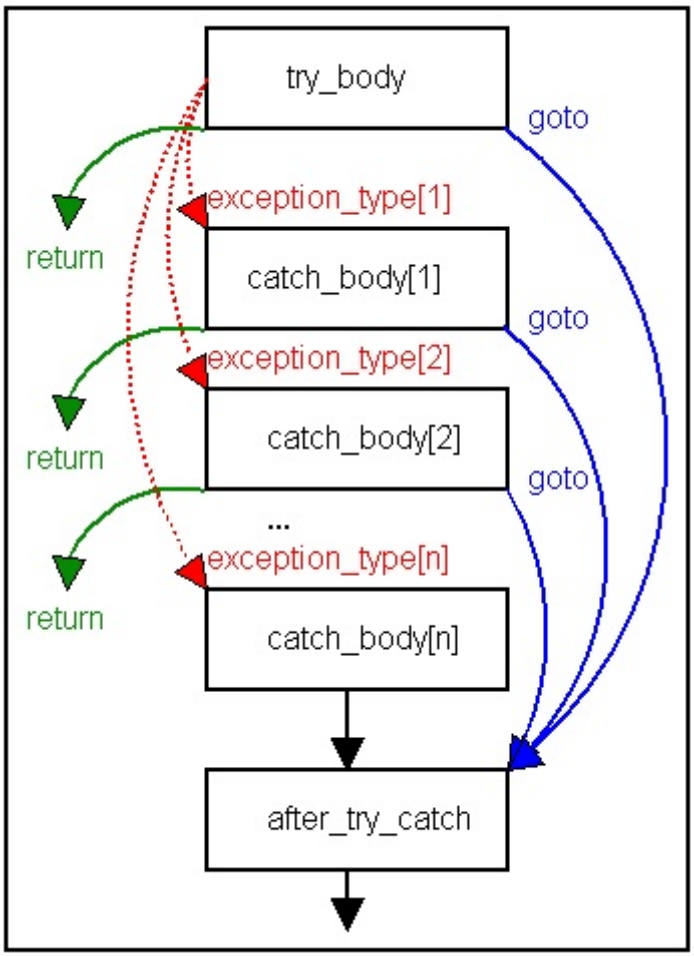

Figure 2.15: Try-catch block.

Chapter 3

Implementation of the library

In this chapter I describe, how the classes described in chapter 2 are implemented. Like in the previous chapter, I also distinguish between the lower (Section 3.1) and the higher abstraction level (Section 3.2).

3.1 Lower abstraction level

Most parts of the implementation of the lower level of the library are simple. It is a straightforward mapping of structures in the Java class file to attributes and their set procedures in the library classes, representing the structures.

As an example of this mapping we look at the class `CONSTANT_NAME_AND_TYPE` from the cluster `constant_pool` that represents the `CONSTANT_NameAndType_info` structure of the Java class file (Table 3.1).

The tag of a name and type constant always has the value 12; therefore, the value of the `tag` feature of class `CONSTANT_NAME_AND_TYPE` is set to a constant of the class `JAVA_CONSTANTS` that has the value 12 too.

The items `name_index` and `descriptor_index` of the `CONSTANT_NameAndType_info` structure, which both hold a constant pool index of a UTF-8 constant, are mapped to attributes with the same name. The type of these attributes is `CONSTANT_UTF8`, which represents a UTF-8 constant in the constant pool. These references are replaced by the indices of the constants when the content of `CONSTANT_NAME_AND_TYPE` is written to the class file. With the set procedures `set_descriptor_index` and `set_name_index` it is possible to change the values of the two attributes.

As I have mentioned before, nearly the whole lower abstraction level is implemented with the same structure as seen in the class `CONSTANT_NAME_AND_TYPE`. But there are also a few points that contain a more sophisticated logic that will be explained in more detail in the following subsections.

3.1.1 Dump a component to the class file

Every class that represents a part of the Java class file inherits from `CLASS_FILE_COMPONENT`; thus it also inherits the deferred feature `dump_component`, which needs to be effected.

The command `dump_component` has a precondition `is_dump_allowed`, which checks whether the required information for dumping to a class file is present. For example, you

Definition	Implementation

CONSTANT_NameAndType_info {
u1 tag;
u2 name_index;
u2 descriptor_index;
}
	class
	CONSTANT_NAME_AND_TYPE
	feature -- Access
	tag: INTEGER is
	do
	Result := const.constant_name_and_type_tag
	end
	descriptor_index: CONSTANT_UTF8
	name_index: CONSTANT_UTF8
	feature -- Element change
	set_descriptor_index (i: like descriptor_index) is
	do
	descriptor_index := i
	end
	set_name_index (i: like name_index) is
	do
	name_index := i
	end
	end

Table 3.1: Name and type constant implementation.

may want to check that the file given as argument is not void. Descendants of `CLASS_FILE_COMPONENT` can redefine `is_dump_allowed` to specify their claims to be able to write their content to the file.

The command `dump_component` only writes to the class file elements that belong to the component itself. If a component contains other components it calls their `dump_component` feature so that they can write themselves to the file.

An example of this behavior is the feature `dump_component` of the class `METHOD` (Figure 3.1). First it calls `dump_component` on the access flags to write them to the class file. Then it writes its name and the descriptor index directly to the file. At the end, if the attribute `attributes` is not void it calls the feature `dump_compnent` on `attributes`. Otherwise, if the method does not have any attributes it writes 0 to the class file (for the size of the attribute table).

The order of the different instructions is also important in this feature because they must be executed in the same order as the data has to be written to the class file. In the method structure of the class file (Figure 2.4), the access flags are in first position, then the name and the descriptor index and at the end the attributes, if there are any.

```
dump_component (file:  JAVA_CLASS_FILE) is
        -- Write the content of the component to the 'file'.
    do
        access_flags.dump_component (file)
        file.put_unsigned_short (name_index.index)
        file.put_unsigned_short (descriptor_index.index)
        if attributes /= Void then
            attributes.dump_component (file)
        else
            file.put_unsigned_short (0)
        end
    end
```

Figure 3.1: Feature dump_component of the class METHOD.

Java class file

To write data to a Java class file there is the class `JAVA_CLASS_FILE` that inherits from `RAW_FILE`. To guarantee the correct order of the bits (big-endian) in multiple byte values, the features of this class are implemented with pointers. This means that they write byte by byte to the class file.

3.1.2 Access flags

The class `ACCESS_FLAGS` and its descendants that are used to manage the access flags of the classes, methods, fields and inner classes use the same encoding for the flags like it is used in the Java class file. This means that the attribute `access_flags` stores exactly the same bitmap as written to the class file.

To make the user's life simple, the class `JAVA_CONSTANTS` defines constants for all access flags that can appear in a class file. (The bit-operations & and | are used to manipulates the flags.)

3.1.3 Constant pool

The cluster `constant_pool` also contains some points in the implementation that need a further explanation. On one hand, this is the computation of the indices of the constant pool entries and on the other hand this is the handling of unicode strings in the UTF-8 constant.

Index computation

To be able to compute the correct indices of the constants in the constant pool there are several things done in the classes `CONSTANT` and `CONSTANT_POOL`.

The attribute `index` represents the index of the constants, defined in the class `CONSTANT`, which is a precursor of all classes that represent a constant pool entry. The set procedure

for this attribute (`set_index`) is only exported to the class `CONSTANT_POOL` because the constant pool is the only instance in the whole library that has a complete overview on all constants. The feature `size` of the class `CONSTANT` returns the size of the constant in the constant pool[1].

The class `CONSTANT_POOL`, which inherits from `CLASS_FILE_TABLE` exports all features that can be used to manipulate the content of the table to `NONE`. The only two exceptions are the features `force` and `extend`, which are redefined. These new versions set the index to a newly inserted constant and compute the new size of the constant pool, using the size of the constant. Then the old version (Precursor) of `force` / `extend` is used to add the constant to the constant pool.

Unicode strings

All strings in the Java class file are encoded in Unicode format UTF-8, but the current version of Eiffel does not support Unicode. Nevertheless, to be able to take unicode strings as input and to generate UTF-8 strings as output, the following two solutions are implemented in the class `CONSTANT_UTF8` (depending on the given input string, you may want to use one or the other):

ASCII input To add a string stored in an object of type `STRING` (encoded in ASCII) to a `CONSTANT_UTF8` one has to use the feature `set_value` and pass the string as argument.

The feature `set_value` first sets the attribute `value` to the passed string and then computes the UTF-8 representation of the string. This conversion is implemented in the feature `value_to_utf8` (exported to `NONE`) that takes every character like it would be a unicode character and computes its UTF-8 encoding. The generated UTF-8 string is stored in the secret attribute `utf8_string` of type `STRING`. A UTF-8 character that is longer than one character also occupies more than one character in the `utf8_string`. The UTF-8 string does not make any sense as an Eiffel string, but if it is written byte by byte to the class file, it builds a correct, UTF-8 encoded string.

Unicode input To add a Unicode string to a `CONSTANT_UTF8` one first has to convert the Unicode characters to `INTEGER_16` values and put them in a `ARRAYED_LIST [INTEGER_16]`. Then one has to call the feature `set_value_unicode` of class `CONSTANT_UTF8` and pass the `ARRAYED_LIST [INTEGER_16]` as argument. The Unicode string is stored in the attribute `value_unicode` and the secret feature `value_unicode_to_utf8` is called: this feature translates the string to UTF-8 encoding and stores the value in the secret attribute `utf8_string`. If the feature `dump_component` of `CONSTANT_UTF8` is called, it writes the content of `utf8_string` byte by byte to the class file.

3.1.4 Instructions

The cluster `instructions` also makes some automation for the user by computing the addresses of the instructions and offsets for the branch and jump instructions. I will now explain these points of implementation of the classes `INSTRUCTION` and `INSTRUCTION_LIST`.

[1] All constant have size 1, except the long and the double constant that have size 2.

Several parts of the class INSTRUCTION are involved in the address computation. The most important part for the user is the attribute address where the computed address can be accessed. Its corresponding set procedure set_address is made selectively available to INSTRUCTION_LIST because this class is the only one that has a complete overview over all instructions of a method and thus it is also the only one that is able to compute the addresses.

But to compute the instruction addresses, the class INSTRUCTION_LIST needs to know the length of every instruction. This information can be obtained by using the query length of the class INSTRUCTION. For most instructions, the length only depends on their opcode, but for the instructions tableswitch, lookupswitch and wide the length can vary because their argument number and argument types are not always the same. Therefore the length computation also needs to examine their arguments.

The class INSTRUCTION_LIST inherits from LINKED_LIST. It redefines all features that can be used to change the content of the linked list. The new implementations of these features manipulate the linked list by using the Precursor versions and then they also update the addresses of the instructions where it is necessary. To simplify these updates there is the secret feature update_addresses that updates all addresses from the current cursor position to the end of the linked list.

The class INSTRUCTION also provides other computations non-related to the address computation. One is in the feature local_variable_use that returns the greatest local variable index that the instruction uses. This feature can be used to compute the maximal use of local variables that has to be set in the code attribute. The query stack_change can be used to compute the maximal stack size of a method because it returns how the instruction changes the stack. This computation is quite sophisticated because the result does not only depend on the opcode of the instruction. The arguments of the instruction also have to be analyzed because for example for the instruction invokevirtual all parameters passed to the invoked method have to be stored on the stack. Therefore the stack change is dependent on the signature of the called method. To compute the desired stack size of the method parameters there are the secret features args_stack_length, class_name_length and array_length.

3.1.5 Java constants

The class JAVA_CONSTANTS mainly contains definitions of constants that are used in a Java class file. The only implementation parts are the features valid_class_flags, valid_field_flags, valid_inner_class_flags and valid_method_flags that are used to check the validity of access flags.

The class JAVA_CONSTANTS inherits from SINGLETON to implement the singleton pattern [GHJV94] [AB]. The singleton pattern is used to avoid multiple instantiation of the class JAVA_CONSTANTS.

3.2 Higher abstraction level

The implementation of the higher abstraction level is much more interesting than the implementation of the lower abstraction level (Section 3.1) because the goal of the higher

level is to let the user specify the important information and then generate the corresponding structures of the lower level automatically.

As mentioned in the description of the library design (Chapter 2) the higher level is divided into two parts. The implementation description will follow the same division. First I will explain the implementation of the structure generators that automate the creation of class file structures (Section 3.2.1). Second I will describe the implementation of the code generators that simplify the creation of Java bytecodes (Section 3.2.2).

3.2.1 Structure generators

The constant pool generator has a very central role in the generators because all other generators for classes, methods, fields and attributes need access to a constant pool to reference items there. This is the reason why I first describe the implementation of the constant pool generator, and then move to the attribute generator and at the end, the class, method and field generators.

Constant pool generator

When creating a constant pool generator one can choose between two different possibilities. Either the creation procedure creates a new constant pool (`make`) or an already existing constant pool is passed as argument to the creation procedure (`make_from_constant_pool`) to be managed by the generator.

The goal of the constant pool generator is to simplify the creation of a constant pool entry and to avoid the appearance of multiple constants with the same content. This is done by either creating a new constant if there is none with the same content, or returning a reference to the existing constant.

Regarding the implementation there are three features for every constant type: (1) The first has the same name as the constant (for example `constant_field_ref`) and takes as argument all information defining the content of the constant. This feature is called if the user wants a reference to such a constant with the given content. The feature checks if there is already such a constant in the pool (using feature `impl_<constant_name>`, see below). If it finds one, it returns its reference, otherwise it creates a new constant, adds it to the pool and returns its reference. (2) The name of the second feature type starts with the prefix 'has_' (for example `has_constant_field_ref`) and also takes as argument the information defining the content of the constant. It checks if there is a constant with the given content in the pool. It is implemented by obtaining a reference to the wanted constant (using feature `impl_<constant_name>`, see below) and then checking if it is void or not. (3) The features of the third type are exported to `NONE` and have the name `impl_<constant_name>` (for example `impl_constant_field_ref`). The implementation iterates over the constant pool and checks for every item if it has the wanted constant type and if it has the same content as the one specified by the feature arguments. If it finds such a constant, it returns a reference, otherwise it returns void.

Attribute generator

The attribute generator does not contain a sophisticated logic, but simplifies the creation of attributes for the users. For every attribute that is described in the JVM Specification [LY99] the class `ATTRIBUTE_GENERATOR` has a feature with the same name. These features

create an instance of the class that represents the wanted attribute. Then, using the constant pool generator of the class, one can obtain a reference to a UTF-8 constant and set it to the `name_index` of the attribute. A reference to this attribute is returned to the user.

Class generator

When creating a class generator the user can choose if he wants to use an already created constant pool (`make_from_constant_pool`) or if the class generator should create a new one (`make`). In both cases, the creation procedure creates a new object of type `JAVA_CLASS` and stores it in the attribute `java_class`. Then a constant pool generator is created and the class is initialized by setting the constant pool and the class name (passed as argument to the creation procedure). A new `CLASS_FLAGS` object is also created and added to the java class. At the end of the creation procedure, a `JAVA_CONSTANTS` object and the attribute generator, used in the class generator, are created.

The 'Element change' features contain several patterns for implementing the class generator. The features `set_major_version` and `set_minor_version` directly call the same features of the java class to set its version numbers. The feature `set_access_flags` does the same to manipulate the access flags of the class.

The feature `set_super_class` has a slightly different implementation. To set the super class of a java class one needs a UTF-8 constant containing the name of the super class. Therefore `set_super_class` first creates this constant and then adds it to the java class.

The features `add_field`, `add_method` and `attribute` first check if the corresponding table in the java class is still void. If it is the case, they create a new table and add it to the java class. Then the field, method or attribute is added to the table.

The feature `add_interface` combines two of the patterns described before. First it checks if the interface table of the java class is void and creates a new one if it is necessary. Then, it creates a UTF-8 constant with the interface name as content, using the constant pool generator. A reference to this constant is added to the interface table of the java class.

```
aload_0
invokespecial 'superclass' <init> ()V
return
```

Figure 3.2: Empty constructor.

A somewhat special case in the class `CLASS_GENERATOR` is the feature `add_empty_constructor`. Using a method generator, this feature creates an empty constructor with the given access flags. The bytecodes of an empty constructor (Figure 3.2) only call the constructor of the super class by first loading a reference to the current object, invoking the constructor on it and returning the control flow to the caller of the constructor.

Method and field generator

The implementation of the classes METHOD_GENERATOR and FIELD_GENERATOR have quite similar structures. The creation procedures that take as argument either a constant pool (make_from_constant_pool) or a constant pool generator (make_from_constant_pool_generator) create a new method or field and initialize them with empty access flags.

The features add_attribute, set_access_flags, set_name and set_descriptor of both classes and add_exception of the class METHOD_GENERATOR are all implemented with the patterns described in the section above about the class generator.

The only exception is the feature add_code of the class METHOD_GENERATOR that is used to add the code attribute to the method. Before adding the code attribute to the method using the feature add_attribute, this feature computes the number of local variables that are needed to store all parameters passed to the method. If this number is greater than the value of max_locals of the instruction table in the code attribute then it replaces the value in the instruction list. This is done to make sure that all parameters can be stored in the local variable even if they are never accessed by the code. The computation of the number of used local variables for the parameters is done by the secret features parameter_size, class_name_length and array_length.

3.2.2 Code generators

The code generators build a block structure that can be recursively combined. Every block can determine the maximal local variable index that occurs in its code and compute the maximal stack depth and the stack depth at the end of the block (both in relation to the stack depth at the beginning of the block). By combining the blocks recursively, all these values also have to be combined. I will now explain how this can be done.

The second challenge are the different tables like the exception table, the line number table and the local variable table that contain extra information about the code and that also have to be put together if two blocks are combined.

Block

The deferred class BLOCK is the ancestor of all other blocks defined in the cluster code_blocks. Even if it is deferred, it contains some parts of the implementation that can be used in all other blocks.

The features max_stack and end_stack check if the values are set by the user (is_max_stack_user_set, is_end_stack_user_set) and then return either the user-set values (user_max_stack, user_end_stack) or the computed values (max_stack_computed, end_stack_computed).

The set procedures set_max_stack and set_end_stack are used to define values set by the user that are stored in the secret attributes user_max_stack and user_end_stack. Of course they also set the value of the 'Status report' features is_max_stack_user_set or is_end_stack_user_set to true. To reset these boolean values and to force the block to compute its values one can use the procedures compute_max_stack and compute_end_stack.

The feature code_attribute creates a code attribute from the whole content of the block. It sets its code (creating an instruction list to compute the instruction indices),

the maximal number of local variables, the maximal stack depth and also the exception table, the line number table and the local variable table if they are not void.

Code block

The code block (class `CODE_BLOCK`) is used to compose instructions of linear code. It means that this section does not contain any jump or branch instructions. This restriction is to keep the computation of the stack depth simple. This is guarded by the feature `is_jump_instruction` that is called from the query `is_complete`.

The class `CODE_BLOCK` inherits from `LINKED_LIST [INSTRUCTION]` and thus the instructions can be added and manipulated by the features of `LINKED_LIST`. Also the queries `first_instruction` and `last_instruction` are implemented with the queries `first` and `last` from `LINKED_LIST`. The query `instructions` returns the code block object itself.

The implementation of `max_local` simply iterates over all instructions in the list and looks for the maximal local variable index that is accessed. This value is returned.

A similar strategy is also used for the stack computations `max_stack_computed` and `end_stack_computed`. Both iterate over the list of instructions and sum up the stack changes of the instructions. `end_stack_computed` returns the final value of this sum and `max_stack_computed` returns the maximal value the sum has reached during its computation. This simple strategy is only possible because no jump or branch operations are allowed.

Handling of tables in this block is also quite simple because it contains no other blocks. The attribute `exceptions_table` is always void because exception attributes can only be created in try-catch blocks (see below). The `line_number_table` and `local_variable_table` are managed by the features `add_line_number_table` and `add_local_variable_table`. If the table is still void when one wants to add an item, a table object must be created before the item can be added.

The command `return_block` is used to set the attribute `is_return_block` to true, to tell that this block always returns to the caller.

Concat block

The concat block (class `CONCAT_BLOCK`) is used to concatenate two blocks stored in the attributes `first_block` and `last_block`. To compute information of the concat blocks it has to combine the information of the blocks it contains.

For the features `first_instruction` and `last_instruction` this is very simple. The first instruction of the concat block is the first instruction of the first block; the last instruction of the concat block is the last instruction of the second block. The implementation of `instructions` first creates a new `LINKED_LIST [INSTRUCTION]` and then appends the instructions of the first block before appending the instructions of the second block.

The query `max_locals` returns the maximum of the `max_locals` values of the first and the second block.

The stack computations (`max_stack_computed` and `end_stack_computed`) also use the values of the appended blocks to compute the new ones. `end_stack_computed` simply adds the end stack values of the two blocks. The implementation of `max_stack_computed` returns the maximum of `max_stack` of the first block and the sum of `end_stack` of the

first and `max_stack` of the second block. The sum of the second block has to be taken into account because the maximal stack depth is always computed in reference to the stack size at the beginning of the block.

The table management is implemented in the following way: To add line numbers or local variable table entries one has to use the features `add_line_number_info` and `add_local_variable_info` that extend the tables of secret attributes `private_line_number_table` and `private_local_number_table` after creating a new table object if necessary. The features `exception_table`, `line_number_table` and `local_variable_table` first create a new result table object. Then they append the content of the corresponding table of the first and the second block if it is not void. For the line number and the local variable table the private tables are also appended if they are not void. If the result table is still empty, which means that there is no information to pass, the result table is set to void.

The implementation of `is_return_block` is quite simple. It returns the same result as `is_return_block` of the second block.

Do-while block

The do-while block (class `DO_WHILE_BLOCK`), used to create a loop, contains only one block for the code inside the loop, which is stored in the attribute `loop_body`.

The first instruction of the do-while block is the first instruction of the loop body and the last instruction of the do-while loop is the branch operation (`test_branch`), which is appended automatically. The implementation of `instructions` creates a new linked list, inserts the code of the loop body and then appends the branch instruction defined in the attribute `test_branch`. The target of the branch instruction is set to the first instruction of the loop body.

The query `max_locals` can return the value from `max_locals` of the loop body because the branch instructions do not access any local variables.

The computation for the end stack depth always returns 0 because the stack size at a certain point in the code always has the same size, independently of the execution path from where it is reached. Thus, the stack depth at the end of a loop must be the same as at the beginning of the loop, otherwise, after every iteration there would be another stack depth at the end of the loop. The maximal stack depth is the same as the maximal stack depth of the loop body because all branch instructions consume more stack items than they produce.

The table handling is quite similar to the one in the concat block (see before). If there are any entries in the loop body, they also appear in the do-while block. For the line number and the local variable table it is also possible to add new entries (`add_line_number_info`, `add_local_variable_info`) that are appended at the end of the tables.

`is_return_block` is constantly set to false, because it does not make any sense for a loop body if all its execution paths return to the caller. There would never be several iterations of the loop.

If-then block

The if-then block (class `IF_THEN_BLOCK`) contains two blocks that are stored in the attributes `then_body` and `after_if`.

The first instruction is always the branch instruction that is stored in the attribute `test_branch` and the last instruction is the last instruction of the after-if block. The implementation of `instructions` first sets the target of the branch instruction to the first instruction of the after-if block. Then it composes the linked list by adding, first the branch instruction, then the instructions of the then-body and at the end the instructions of the after-if block.

The feature `max_locals` simply returns the maximum of the `max_locals` queries of the then-body and the after-if block. The branch instruction can be ignored for this computation because the branch instructions do not access any local variables.

The computed stack depth at the end of the block is the sum of the stack change of the branch instruction and the after-if block. The then-body block can be ignored because the stack size at a certain point must always be the same, independently of the execution path. For the maximal stack depth, the maximum of the following three values has to be taken:

- stack change of the branch instruction
- sum of the stack change of the branch instruction and the maximal stack depth of the then-body block
- sum of the stack change of the branch instruction and the maximal stack depth of the after-if block

The table handling is implemented by combining the tables of the then-body, the tables of the after-if block and for the line number and the local variable table the tables created in the if-then block.

The if-then block is a return block (`is_return_block`) if the after-if block is a return block.

If-then-else block

The if-then-else block (class `IF_THEN_ELSE_BLOCK`) looks quite similar to the if-then block, but there are several differences in the implementation. It contains the three blocks `then_body`, `else_body` and `after_if`.

The first instruction is the branch instruction (`test_branch`). The last instruction depends on the then- and the else-body block. If they are return blocks, the if-then-else block does not need an after-if block and the last instruction is the last instruction of the else-body block. Otherwise, the last instruction is the last instruction of the after-if block. The implementation of `instructions` first sets the target of the branch instruction to the first instruction of the else-body block and then adds the branch instruction (`test_branch`) and the then-body to the list. Then, if the then-body is not a return block, a goto instruction with the first instruction of after-if as target must be added. The else-body follows the after-if block, but only if the then-body and the else-body block are not return blocks.

The computation of the maximal number of used local variables (`max_locals`) determines the maximum of the `max_locals` from the then-body and the else-body and, if then- and else-body are not return blocks, of the after-if block.

The sum of the stack change from the branch instruction and the then-body block is calculated to compute the stack change (`end_stack`) of the if-then-else block. If then-

and else-body are not return blocks, the stack change of the after-if block must also be added.

The if-then-else block is a return block if either the then-body and the else-body are return blocks or if the after-if block is a return block.

Print block

The implementation of the print block (`PRINT_BLOCK`) is quite simple because it does not contain any other blocks.

The first instruction is always a `getstatic` instruction to load the `PrintStream` object that writes to the standard output and the last instruction is always an `invokevirtual` instruction that invokes the method `print` or `println` of the loaded `PrintStream` object. The implementation of the query `instructions` distinguishes between three cases. In the first case, only a line break is written and the method `println` does not take any arguments that have to be loaded. In the second case, an eight byte value has to be written. To swap its position with the reference to the PrintStream object the instructions `dup_x2` and `pop` have to be used. The third case writes a four byte value to the standard output. To swap the stack positions of the value and the reference to the PrintStream, the swap instruction is used.

The number of used local variable is always zero because none of the used instructions accesses a local variable.

The maximal stack depth becomes 2 if an eight-byte value is written, otherwise it is 1. The computation of the end stack distinguishes between three cases: If only a line break is written, the stack change is 0, if an eight byte value is written the stack change is -2, else the stack change is -1.

The exception table is always void and items for the line number and the local variable tables can be added with the features `add_line_nubmer_info` and `add_local_variable_info`.

The print block is never a return block.

Switch block

The switch block (class `SWITCH_BLOCK`) contains a certain number of case blocks and sometimes an after-switch block. The case blocks (`case_blocks`) are stored in an `ARRAYED_LIST [TUPLE [INTEGER, BOOLEAN, BLOCK, BOOLEAN]]` where the integer stores the value of the block, the first boolean tells if it is the default tag, the block corresponds to the case block to be stored and the last boolean tells if the block contains a break. If there is a default block and either all case blocks are return blocks or no case block contains a break then the after-switch block does not have to be defined.

The first instruction of a switch block is always the switch instruction defined in `switch_instruction`. If the after-switch block is not void, the last instruction is the last instruction of the after-switch block. Otherwise, the list of case blocks has to be iterated in the reversal order and the last instruction of the first block found must be taken. If there is no block in the list, the switch instruction is also the last instruction.

The generation of the instruction list is quite a sophisticated thing. First an iteration over the case blocks is proceeded to find the target for the default jump. If there is no default block in the case blocks, the first instruction of the after-switch block is the target

of the default jump. Then one must distinguish between the two switch instructions `tableswitch` and `lookupswitch`. For the `tableswitch`, an array for the range of the values is created and all entries initialized to the default jump. Then there is once again a loop that iterates in the reversal order over the case blocks and sets all defined jump targets in the table. At the end, the table is set as parameter of the `tableswitch` operation. For the `lookupswitch` instruction, a table of `MATCH_OFFSET` objects sorted to the match entries has to be built up and added as parameter to the `lookupswitch` operation. The created switch instruction (either `tableswitch` or `lookupswitch`) must be added to the resulting instruction table followed by the code blocks of the case blocks table. If a code block contains a break, a goto instruction that jumps to the after-switch block must be added. At the end, the instructions of the after-switch block have to be added, if present.

The computation of the `max_locals` value builds the maximum of the `max_locals` from all case blocks and, if present, of the after-switch block.

The stack computations are quite simple. The `end_stack` value is the end stack value of the first block in the case blocks, added with the end stack value of the after switch block, if present. The `max_stack` value is the maximum of all `max_stack` values of the case blocks and the value that results from the computation '`end_stack_computed - after_switch.end_stack + after_switch.max_stack`' if there is an after-switch block.

The tables are created by appending the tables of the containing blocks in their order of appearance. For the line number and the local variable table it is possible to append new items at the end with the features `add_line_number_info` and `add_local_variable_info`.

The switch block is a return block if either the after-switch block is a return block or if the after-switch block is void and all case blocks are return blocks.

Try-catch block

The try-catch block (class `TRY_CATCH_BLOCK`) may contain three different types of blocks. At the beginning comes the try-body, then some catch-bodies (stored in `exception_handlers`) and at the end the after-try-catch block. If the try-body and all catch-bodies are return blocks then the after-try-catch block does not have to be defined.

The first instruction is always the first instruction of the try-body. The last instruction is either the last instruction of the after-try-catch block or if there is no after-try-catch block the last instruction of the last catch-body. The implementation of the query `instructions` first appends the instructions of the try-body to the list. If the try-body is not a return block, a goto instruction with the first instruction of the after-try-catch block is added as target. Then the instructions of the catch-bodies are appended and if a catch-body is not a return block, a goto has to be added. At the end, the after-try-catch block is appended if it is present.

The `max_locals` value is the maximum of the `max_locals` values of all involved blocks.

The value `end_stack` is computed by adding the stack change of the try-body with the stack change of the after-try-catch block. If the after-try-catch block does not exist, the `end_stack` is computed by adding the stack change of the try-body and the stack change of the first catch-body block plus 1 (exception type is passed on the stack). The maximum stack depth is the maximum of the following values:

- `max_stack` of the try-body

- the maximum of `max_stack` of a catch body added with `end_stack` of the try-body
- the sum of `end_stack` of the try-body and `max_stack` of the after-try-catch block, if present

The query `exception_table` is implemented by adding for every catch-body an exception info entry to the table. If one of the blocks contains additional exception table entries, it is added at the end of the table. For the line number and the local variable table, the tables of all involved blocks are concatenated; items added with the features `add_line_number_info` and `add_local_variable_info` are added at the end of the tables.

The try-catch block is a return block if the try-body and all catch-bodies are return blocks or if the after-try-catch block is a return block.

While block

The while block (class `WHILE_BLOCK`) contains two blocks, stored in the attributes `loop_body` and `after_loop`.

The first instruction is set to the branch instruction that is defined in the attribute `test_branch`. The last instruction is set to the last instruction of the after-loop block. The instruction list of the query `instructions` is generated by first setting the jump target of the branch instruction to the first instruction of the after-loop block and then adding the branch instruction to the table. After the instructions of the loop-body, a goto is added; it jumps back to the branch instruction. At the end, the instructions of the after-loop block are appended.

The query `max_locals` returns the maximum of the `max_locals` values of the loop-body and the after-loop block.

The `end_stack_computed` value is the sum of the stack change of the branch instruction and the `end_stack` value of the after-loop block. The maximal stack depth is the maximum of the stack change of the branch instruction, the sum of the stack change of the branch instruction and the maximal stack depth of the loop-body block and the sum of the stack change of the branch instruction and the maximal stack depth of the after-loop block.

The tables are handled by concatenating the tables of the loop-body and the after-loop block. For the line number table and the local variable table, the items added with the features `add_line_number_info` and `add_local_variable_info` are appended at the end of the tables.

The while block is a return block if the after-loop block is a return block.

Chapter 4

Test language and compiler

The library described in this report (Chapters 2 and 3) can be used in a back-end of a compiler whose target language is Java bytecodes.

I decided to write a compiler that uses the library in the back-end for the following reasons:

- I can give an example of how the library can be used in a back-end of a compiler.
- A compiler based on the library that jumps to the after-switch block must be added.
- It is possible to estimate the usability of the library in a back-end for a compiler. It is possible to find problems and to suggest ways to solve them.

First I want to write a few words about why I decided to choose the Java programming language and which parts of the language I have really implemented (Section 4.1). In another section I write about the compiler and how it is built up (Section 4.2).

4.1 Language

In a first thought about the choice of the source language I had three possibilities in my mind: Eiffel, Java or define an own language. Of course I could have thought of every other programming language but I chose these three for the following reasons:

Eiffel is my implementation language and the library will be used to create a compiler that translates Eiffel to Java bytecodes. But the structure of Eiffel has too many differences to the bytecodes structure (for example multiple inheritance) so that I would not have enough time to finish this project during my diploma thesis.

The advantage of defining my own language is certainly that I can keep it quite simple but then it will not be very powerful. Another problem is that I have to write my own unique grammar that has to be accepted by a parser.

The structure of the Java programming language is very close to the structure of the bytecodes because they are designed to be used together. But Java is, like Eiffel, an extensive programming language and it can become too much for my project.

Thus, I decided to use the grammar of the Java programming language like it is defined in the Java language specification [GJS96], but the compiler will only be implemented for certain parts of the language.

The next sections describe which parts are implemented.

4.1.1 Expressions

Expressions define the basic functionality of the Java programming language. Much of work in a program is done by evaluating expressions. They start the execution of a statement sequence (method invocation), return values (computations, resolving names) and can also produce side effects (assignments to variables).

In my compiler, the following expressions are implemented and can be compiled:

Assignment The assignment expression is implemented, but only for the operators =, += and -=. = is the normal assignment operator and `a += b` respectively `a -= b` are short versions for `a = a + b` and `a = a - b`. The return value of an assignment expression is the value that is assigned to the left-hand side.

Binary operation There exists several binary operations for different data types. The compiler can handle the following ones: The numeric operators *, /, %, +, - and the comparison operators <, >, <=, >= for all numeric data types (integer, long, float, double), the logical operators & (and), | (or), ∧ (xor) for integral data types (integer, long) and boolean, && (conditional and) and || (conditional or) also for booleans and the `instanceof` operator for objects.

Cast expression The cast expression is also implemented in my compiler, but only if the expression in the parenthesis is a class name.

Method invocation It is possible to invoke methods of the current class or in other objects (specified by a object reference). If the compiler cannot resolve the signature of the method, it asks the user to give additional information.

Class instance creation The creation of new class instances by calling the constructor is also implemented.

Literal expression Literals in the code are recognized and their value can be loaded on the stack.

Name expression Names can also be resolved. The compiler recognizes local variables and fields that are defined in the current class. For all other names it asks the user for the type of the name to check if it is a class name used to access a static field or method.

4.1.2 Blocks and statements

The blocks and the statements are used to structure a program and to define its execution order. A block is a container for statements and also builds a scope in which a local variable can be defined. The statements define the control flow of the programs by constructs like loops, if-then-else, switch or exception handling.

In my compiler, of course the block construct is implemented because without blocks it is not possible to compile any program. For every block there is a new scope for the local variables defined.

The statements that can be compiled with the compiler are the followings:

Local variable declaration statement The local variable declaration statement can be used to declare one or more local variables in a block. It is also possible to initialize these variables in the local variable declaration statement.

Expression statement Some expressions in the Java programming language can also be used as statements. These are the assignment, the method invocation, and the expression to create a new class instance. The difference if an expression is used as statement is that it does not have a return value that is stored on the operand stack. In the Java programming language it is also possible to use the increment respectively decrement expressions (++ and --) as statements, but in my compiler the compilation of these expressions is not implemented, hence it is not possible to use them as statements either.

If-then-else statement The if-then-else statement is also implemented in my compiler. The evaluation of the boolean expression that decides which code is executed writes always a 0 (false) or a 1 (true) onto the stack. This number is examined by the branch instruction of the if-then-else statement.

Switch statement The switch instruction is implemented using the switch block. The use of the break statement is not implemented like it is described in the Java language specification. In my compiler it is only checked if there is a break in a case block. If one is found, the program jumps at the end of the case block (even if the break is not at the end) to the code after the switch statement.

Do-while loop statement The do-while loop can also be compiled with my compiler. The boolean expression to test whether another iteration has to be taken, writes always a 0 (false) or a 1 (true) onto the stack that is examined by the branch instruction of the loop statement.

Try-catch statement The try-catch statement, used to handle exceptions can also be used with my compiler but it is not possible to access the exception object in the catch clauses. The finally statement cannot be used either.

4.1.3 Packages, classes and interfaces

Until now, in this section I have only written about expressions, statements and blocks that define the logic and the control flow of single program parts. But to really be executed, a program also needs an environment. In the Java programming language these are the methods that contain the code and that themselves are parts of classes. Of course these constructs also have to be compiled, but again, at this level I have not implemented everything in my compiler. It supports the following structures:

Compilation unit The compilation unit is everything that is stored in a Java file. It can contain the package declaration, the import declarations and type declarations. My compiler ignores package and import declarations. It only handles some type declarations (see below).

Class declaration The class declaration contains all information that is needed to build up a class. It is supported by my compiler that writes for every class one class file

to the disk. Interfaces are not defined with the class declaration and they are also not supported by the compiler.

Field declaration Fields are accepted as contents of the class. If an initial value is added, the initialization is added to all constructors of the class.

Method declaration It is also possible to define and compile methods with my compiler. This information is stored in the method declaration.

Constructor declaration To define a constructor for a class one has to write a constructor declaration. The compiler adds to every constructor first a call to the constructor of the superclass and then the initialization of the fields. Then it adds the constructor code, defined in the Java file.

4.2 Compiler

As described before, I decided to write a compiler that uses parts of the Java programming language as source language and compiles them to Java bytecodes, using the created library (Chapters 2 and 3).

For implementing the compiler I did the following three steps: I first wrote a scanner and a parser for the Java programming language (Section 4.2.1). The parser builds up an abstract syntax tree (AST, Section 4.2.2). Visitor classes traverse the AST to analyze the gained information and to build structures of the library (Section 4.2.3). The library writes its content to a Java class file that can be executed in a Java virtual machine.

4.2.1 Scanner / Parser

For creating the front-end of my compiler I used Gobo Eiffel Lex and Gobo Eiffel Yacc of the Gobo Eiffel library [Bez] to generate a lexical analyzer and a parser. The definition of the lexical structure and the grammar for the Java programming language are taken from the the Java language specification [GJS96].

Lexical structure

In Java, a white space is defined as space, horizontal tab, form feed or a line terminator. In a program they are used to separate tokens, but otherwise they are simply ignored. In my lexical analyzer I have defined a special rule for the line terminators to count the lines in the analyzed program.

In Java there are two kinds of comments: (1) The single line comments that start with `//` and end at the end of the line and (2) the traditional comments that are enclosed by `/*` and `*/`. Comments are ignored in the lexical analysis.

The keywords of Java (Table 4.1) are character sequences that cannot be used as identifiers in a program.

The literals that represent values in a Java program are much more sophisticated to handle in the lexical analyzer than the keywords, because after their recognition, their value has to be computed and passed to the parser. There are several kinds of literals:

abstract	default	if	private	throw
boolean	do	implements	protected	throws
break	double	import	public	transient
byte	else	instanceof	return	try
case	extends	int	short	void
catch	final	interface	static	volatile
char	finally	long	super	while
class	float	native	switch	
const	for	new	synchronized	
continue	goto	package	this	

Table 4.1: Java keywords.

Integer literals There are two types of integer literals, either int (returns an `INTEGER`) or long (returns an `INTEGER_64`) literals. In a Java program they can both be appear in decimal, hexadecimal and octal encoding.

Floating-point literals For the floating-point literals there are also two different types: the float (returns a `REAL`) and the double (returns a `DOUBLE`) literal. There are also several possible encodings, with or without decimal point and with or without decimal power.

Character literals A character literal is always enclosed in ASCII single quotes (’). Because Java supports Unicode, a character literal is always returned as Unicode value (`INTEGER_16`). The encoding of character literals is either the character itself (‘a’), a Unicode escape (‘\u0061’ for a), an octal encoding (‘\141’ for a) or an escape sequence for special characters (‘\t’ for a horizontal tab).

String literals String literals consist of zero or more characters and are enclosed in double quotes (”). The characters of a string have same encodings as the character literals (described before). A string literal returns an array of the Unicode values of its containing characters (`ARRAYED_LIST [INTEGER_16]`).

Boolean literals A boolean literal is very simple. It is either `true` or `false`. It returns a `BOOLEAN` with the corresponding value.

Null literal The simplest literal is the null literal that is `null`. It does not return a value.

Another kind of tokens occurring in a Java program are identifiers. They are built of a letter followed by zero or more letters or digits. If an identifier is detected by the lexical analyzer, a `STRING` is returned with the identifier as content.

The last two types of tokens that are possible in a Java program are the separators (Table 4.2) and the operators (Table 4.3).

(	)	{	}	[	]	;	,	.		

Table 4.2: Java separators.

=	>	<	!	~	?	:				
==	<=	>=	!=	&&	||	++	--			
+	-	*	/	&	|	^	%	<<	>>	>>>
+=	-=	*=	/=	&=	|=	^=	%=	<<=	>>=	>>>=

Table 4.3: Java operators.

Grammar

To be able to use the parser generator Geyacc of the Gobo Eiffel library [Bez] one has to use a grammar for the source language that is context-free and LALR(1). Otherwise it would not always be possible for the compiler to understand the program correctly because it is not always clear how to interpret a certain construct.

The grammar defined in the Java language specification [GJS96] (Appendix B) is already LALR(1); thus it can be taken without any modification to the parser description file that creates a parser for the Java programming language.

4.2.2 Abstract syntax tree

To be able to work on the data retrieved by the lexical analyzer and scanner, it has to be stored in an internal form, an abstract syntax tree (AST). An AST represents the structure of a programming language and allows it to store all data retrieved by a parser.

The AST of my compiler is divided into three levels. There is a level for the expressions, one for the statements and blocks and one for the structures like compilation units, classes, interfaces, methods, fields and constructors. All nodes of the AST inherit from the class `NODE`, which on the one hand is used to implement the visitor pattern (feature `accept`) and on the other hand contains the attribute `line_number` with the corresponding set procedure `set_line_number` that stores the line number of the source file, where the node is created. This information can be used to debug a Java program.

All AST nodes used to represent an expression inherit from the class `EXPRESSION` that defines the deferred feature `type` used to specify the type of the expression value.

The nodes representing statements or blocks are all descendants of the class `BLOCK_STATEMENT` that does not define any features but is used for a structural purpose to define a type for attributes that may store any statement or block.

All other nodes, representing structures of a java program do not have a common ancestor (except `NODE`) because they are too different and there is no structural need for such an ancestor.

4.2.3 Visitors

To be able to create structures of my library from the information stored in the AST, I implemented three visitor classes. This means that the AST is traversed three times. The first traversal gathers information about the methods and fields in the class, the second traversal computes all types of expressions and the third traversal builds up the structures of the library.

Class member visitor

The first visitor (class `CLASS_MEMBER_VISITOR`) only visits some of the structure nodes. It collects the following information:

- name and descriptor of all methods in the class
- name and descriptor of all fields in the class
- initial value of the fields that are initialized

This information is used in the next visitor to resolve names and determine their types. The initial values of the fields are used to write them to all constructors of the class.

Type visitor

The type visitor, as second iteration over the AST, visits nearly all nodes. It is used to determine the types of the expressions. If it cannot resolve a name by checking for a local variable, a parameter or a field of the current class, it asks the user to specify its type.

The user is also asked if the compiler cannot resolve a method call because it does not know this method with the given parameters.

Thus it is possible to invoke every method and to access every field, even the static ones, in the whole Java system, without being able to look at a class, except the currently compiled one.

Code visitor

The code visitor also iterates over the whole AST. By using the information gained in the previous steps (parsing, class member visitor, type visitor) it builds up structures of the higher abstraction level in the library. At the end, the generated `JAVA_CLASS` objects are dumped to class files.

Chapter 5

Discussion

During my project I have gained a lot of experience in designing, implementing and using a Java bytecodes generator and making abstractions. Also the design of an interface that can be used by a compiler is much more difficult and sophisticated than I had thought before and I would do several things different if I could start once again.

To share my experience with other people that will work with this library, I discuss in this chapter the design and the implementation of my library and of the compiler.

5.1 Lower abstraction level

Some parts of the lower abstraction level of the library are not handled in a proper way and the user should be aware of them. There are also two points that I could not solve properly, because the current version of the Eiffel base library, first does not support Unicode (Section 5.1.2), second does not allow to choose between big-endian and little-endian byte order when writing multiple-byte values to a file (Section 5.1.3).

5.1.1 Instructions

In my library, I decided to use only one class (`INSTRUCTION`) to handle bytecode instructions. This class contains all the information and all the algorithms for all possible instructions. Another approach would have been to have one class per instruction (like in BCEL [BCE]). In my view, both approaches have advantages and disadvantages that I will list here, but I do not know which one is really better.

One class for all instructions

One advantage of the approach that puts all instructions in one class is certainly that there is only one class in the library. This makes the class hierarchy much clearer and there is also only one class to be maintained. The user does never have to care about the type of the instructions because they all have the same object type. But it is nevertheless possible to distinguish between different instructions by accessing their opcode feature.

But there are also some disadvantages, especially if we look at the class `INSTRUCTION`. The code is very long (the class has more than 1000 lines of code) and it becomes very difficult to understand everything in such a huge context. There are also very long if-then-else constructs that are used to distinguish between different groups of opcodes.

For every instruction one class

The advantages of the approach that defines for every instruction one class are that the classes are simple and small and they can be optimized for one single instruction. The programmer does not have to write code to distinguish between the different instructions, this is done by the run-time system by taking the implementation of the dynamic type.

The disadvantages are that a lot of classes needs to be created. Not only does every instruction need a class, but there must also be a common ancestor to be able to have lists of instructions for example (i.e. write a `LIST [INSTRUCTION]` rather than declare a `LIST [ANY]` and use assignment attempts everywhere) and several ancestor classes grouping instructions sharing some implementation parts. It can also be more work for the user of the library because he has to distinguish between the different object types (at least for the creation and implementation of the instructions).

5.1.2 Unicode

A problem that occurred during the implementation was that a java program can contain Unicode characters and Unicode strings, but the current version of the EiffelBase library does not support Unicode. Thus I decided to implement a Unicode-tunnelling, which is implemented in the class `CONSTANT_UTF8`.

This means that instead of characters, I use `INTEGER_16` objects, containing the unicode value of a character. These values are converted to the UTF-8 encoding and inserted byte by byte to a string. This string, treated as a usual Eiffel string may contain strange characters that do not make any sense, but if it is written byte by byte to the class file, there will be a correct unicode string encoded in UTF-8.

5.1.3 Java file

Another problem that came up during the implementation was the byte-order in which the values have to be written to the file. In a Java class file, all data longer than one byte (16-bit, 32-bit and 64-bit) have to be divided into two, four or eight bytes and written in big-endian order to the file, where the high byte comes first.

The class `RAW_FILE` of EiffelBase writes data types that are longer than one byte in little-endian order to the file and there is no possibility to change this. Another problem is that `RAW_FILE` does not support all numeric data-types. For example it is not possible to write the value of an `INTEGER_64` object directly to the file.

The solution I adopted to solve these problems is to write my own class `JAVA_CLASS_FILE` that inherits from `RAW_FILE`. In this class I defined put-features for all data-types that have to be written into a class file. The implementation of these features uses the feature `put_data` from `RAW_FILE` that enables writing the data byte by byte and in the desired order to the class file, because the bytes are addressed by pointers.

5.1.4 Access flags

Parts of the classes representing the access flags (cluster `access_flags`) could be designed in a better way. For example the query `valid_flags` in the classes `CLASS_FLAGS`, `FIELD_FLAGS`, `INNER_CLASS_FLAGS` and `METHOD_FLAGS`: The implementation is just a call to a feature of class `JAVA_CONSTANTS` that checks whether the given flags are valid for the

current access flags class. I think, it would be much nicer to have this implementation in the access flags classes. But I had to move the implementation because by using generator classes, it is possible to set access flags values without having a direct access to an access flags object. To make it possible to check the validity of the access flags, I have defined the queries in the class `JAVA_CONSTANTS`.

5.2 Higher abstraction level

Using the higher abstraction level to build small examples (e.g. creating a class file by directly using the generators and the blocks) and especially when using it as back-end for my compiler, I found several things that could have been done better.

I think the generator classes (`CLASS_GENERATOR`, `CONSTANT_POOL_GENERATOR`, `ATTRIBUTE_GENERATOR`, `METHOD_GENERATOR`, `FIELD_GENERATOR`) are quite nice in design and implementation. Their usage is simple and they can be applied in a general way, independently of the source data that has to be mapped to Java bytecodes.

I still think, the idea of using blocks to compose the code of the methods was a good choice, but the implementation of the current version in the cluster `code_blocks` could be much better. In this section I will discuss the problems I have found by using this structure.

5.2.1 General design: Class `BLOCK`

The block structure is defined in the deferred class `BLOCK`, which specifies for all other blocks the functionality they have to implement. In my opinion, this design is good because it provides all information needed to build a code attribute but not more.

The only disadvantage that I can think of is, that every block has to be able to compute its maximal stack depth, this makes it impossible to have jumps or branches in a block that have an instruction in another block as target.

5.2.2 Loop blocks

In my implementations of the loop blocks (`WHILE_BLOCK`, `DO_WHILE_BLOCK`) there is an attribute `test_branch` that stores the branch instruction used for the loop. But this is not enough. To execute a branch instruction in the JVM, the values to be compared must be on the top of the stack. This means that after every loop iteration they have to be loaded and therefore the code that loads these values must be at the end of the loop body. For the while loop, the same code must also be at the end of the block before the while block. But this code does neither belong to the loop body nor to the block before the loop block, it belongs to the branch instruction. It would be better to have an additional block for these instructions that is inserted during the code generation at the places where it has to be.

5.2.3 Concat block

The concat block (class `CONCAT_BLOCK`) is the most commonly used block after the code block because it is the only way to concatenate two blocks linearly. But using this block in

a compiler back-end is not very efficient because a compiler generates lots of code blocks that contain only one or two instructions. To compose them into a linear code section one has to build up a tree of concat blocks. Therefore it would be better if the concat block were a list of blocks that can contain several blocks and even single instructions.

5.2.4 Code block

The code block (class `CODE_BLOCK`) is a list of JVM instructions. It also contains parts that could be improved to make this block more usable. On the one hand the status report attribute `is_return_block` has to be set manually by the user with the procedure `return_block`. It would be better if the code block could analyze its containing code and determine whether it is a return block. On the other hand this is the restriction that no branch or jump instructions may be inserted into a code block. But I did not find a simple solution of how to compute the maximal stack depth of a code block that may contain branch and jump instructions.

5.2.5 After-* blocks

The after-* blocks, which can be found in the blocks `IF_THEN_BLOCK`, `IF_THEN_ELSE_BLOCK`, `WHILE_BLOCK`, `TRY_CATCH_BLOCK` and `SWITCH_BLOCK`, were created to be able to compute the target of branch or jump instructions. But by using these blocks in a compiler, there is no direct possibility to fill these after-* blocks. Thus, in my compiler all after-* blocks contain only a `nop` instruction that does not do anything. It would be better, but also more difficult, to compute the branch and jump targets in a block in a higher level of the block hierarchy and remove the after-* blocks.

5.2.6 Efficiency

The current version of the block implementations and especially the computation of the end and the maximal stack depth are not very efficient. Every time when the features `max_stack_computed` or `end_stack_computed` are called, the values have to be recomputed. If there is a block hierarchy, some values are computed several times because they are needed at several places. I think it would be more efficient if the values are computed only once and then stored in a buffer. In this case, there must also be a flag that indicates if there was a change in the block since the last computation and then force a new computation.

5.2.7 Block structure – source language

The block structure is an intermediate level between the source language and the target language. It is used to build constructs of the source language to map them to the Java bytecodes structures. Every programming language has different constructs. For example Java has three different kinds of loops and Eiffel has only one. Thus, I think it does not make sense to create a general set of blocks that should be used for every possible source language. It is much better to create an interface only for these blocks, like it is done in the class `BLOCK` and then for every source language, the compiler writer builds its own blocks that are optimized for the source language and the AST that its parser creates.

5.3 Compiler

The Java compiler I have created, based on my library, is not a very good compiler because I did not have enough time to create a good design and write a good implementation. But even so, I think it was very valuable for my project because it gave me the possibility to work with the library and gain a lot of experience that allows me to analyze its usability, find the parts that should be improved and also give advice how the library can be used.

If someone wants to write a Java compiler in Eiffel, using my library, I think it would be much more efficient to write a new compiler than using my AST and the visitor classes.

First of all, I have used a very old version of the Java grammar [GJS96], because it was the only Java grammar I could find that was already LALR(1) and therefore compatible to the Gobo Parser Geyacc [Bez]. To create a good compiler one should use the grammar of the latest Java version.

But the biggest problems of my compiler are in the AST (Section 5.3.1) and in the implementation of the visitors (Section 5.3.2).

5.3.1 Abstract syntax tree

My goal was to keep the compiler as simple and small as possible. But this was not a good attitude when creating the AST. I tried to build an AST that has as less classes as possible by mapping similar grammar structures to the same class. At the beginning this looked very nice and I also thought it would be much easier to understand and to work with. But at the end I realized that this is a bad design because the visitor classes have to distinguish between the different structures composed to one class and therefore I had to insert lots of checks to do this.

5.3.2 Visitors

The implementation of the visitor classes could also be much better than it actually is. Of course, the quality of the visitors depends a lot on the AST and there I already had a bad basis. But independently of the AST, I also inserted lots of ugly style elements by using class attributes to pass information from one visit feature to another.

Besides, I have not defined implementation features for functionality that can be used by several visit features, because I did not see at the beginning that it is nearly the same. When I realized it, it would have been too much work to fix it and I was running out of time.

Chapter 6

Outlook

In this chapter I want to give some advice to future users and developers of my library. I will show how the problems mentioned in the discussion (Chapter 5) can be solved, how the library can be used and also where there are possibilities to extend the library.

6.1 Unicode

One of the biggest problems that occur by writing a Java bytecodes generator in Eiffel is that the current version of the Eiffel base does not support Unicode. At the moment, I have solved this problem by storing the Unicode characters in `INTEGER_16` objects. After converting them to the UTF-8 encoding they are stored in a normal Eiffel `STRING` where multiple byte character also occupy multiple bytes in the string. By writing these strings to the class file, they can be treated like normal strings.

In the lower abstraction level of the library, the classes `CONSTANT_UTF8` and `JAVA_CLASS_FILE` are the only classes that are in touch with this Unicode tunnelling. In the higher abstraction level only the class `CONSTANT_POOL_GENERATOR` supports the Unicode tunnelling. When the Eiffel library will support Unicode, the classes `CLASS_GENERATOR`, `FIELD_GENERATOR` and `METHOD_GENERATOR` should also be adapted. Indeed, it is possible in Java to have Unicode class, field and method names.

6.2 Java file

The class `JAVA_CLASS_FILE`, which inherits from `RAW_FILE`, is used to write Java classes into class files. The implementation of the java class file uses the raw file feature `put_data` that writes bytes, addressed by pointers, to the file. This approach is used because it is not possible with the current version of the class `RAW_FILE` of the EiffelBase library to write all numeric types (for example `INTEGER_64`) to a file and multiple byte values are written in little-endian order, but Java needs big-endian order. But this implementation with pointers should be changed, once EiffelBase provides more functionality.

6.3 Wide addresses

Currently the maximal code length in a code attribute of a method is 65535 bytes. Thus it is possible to express every offset of a branch or a jump instruction with two bytes. But

it is possible that future versions of the JVM allow longer code segments. In the current set of JVM instructions there already exist `goto_w` and `jsr_w` that can be used instead of `goto` and `jsr`. The difference is that the instructions ending with '_w' take a four bytes value as offset and the instructions without '_w' take a two bytes value as offset. When a four byte offset becomes possible, it would be nice if the instruction itself decides which instruction should be used, depending on the offset value.

6.4 Instruction factory

Something that could improve the usability of the library would be an instruction factory. I am thinking of a factory that contains for every instruction a feature that takes all the information (parameters) of the instruction as argument and then returns a new instance of this instruction, initialized with the passed arguments.

By using the higher abstraction level of the library, this factory could also be combined with a code block, so that every created instruction would be directly inserted into the block. This would make it much easier for the user. Currently he has to create the instruction, initialize it and then add it to the block. With the factory, he would only have to call one feature with all the information as argument and all the rest would be done automatically.

6.5 Class generator

The class generator could contain additional functionality that would simplify the creation of a class by the user. I propose a data structure that stores the value of this initializer for all class fields that have an initializer. This information can be used to automatically generate a code block that must be appended to all constructors of the class, to initialize the fields like the user defined them.

To make this possible, the initialization of fields also has to be stored in the object representing the field.

6.6 Blocks

As I wrote in the previous chapter, I think it does not make sense to write a general block structure that can be used for every source language. It is much more efficient to create for every source language its own block structure. Here, I only want to give some general advice that should help writing a good block structure.

Even if I propose to write a block structure for every source language, one should not forget the target language. The structure of the blocks should be in a way that it can be easily converted to Java bytecodes. For example it is not sufficient for a loop block to give a special status to the branch instruction because for every branch, there must also be values loaded on the stack. Thus, it would be better to combine the load and the branch instructions to a block and give them a special status.

It is also not good to create blocks that do not have any relation to the source language. I did this mistake with the after-* blocks. I was not able to fill these blocks with reasonable code; hence I had lots of `nop` instructions in my code.

The two most important blocks are the block used to compose linear sections of code (Section 6.6.1) and the block that composes linear sections of blocks (Section 6.6.2). Another very important part of all blocks is the computation of the stack depth (Section 6.6.3).

6.6.1 Code block

Basically a code block is only a list of instructions that also implements the deferred features of the class `BLOCK`. But there is some additional functionality that can be added to make it much more usable in comparison to the code block I have implemented.

It would be very nice if the code block could be combined with an instruction factory that generates instructions and automatically inserts them into the block (Section 6.4).

Another improvement would be that the code block can analyze its own code to determine if all its execution paths return to the caller of the current method, namely the code block is a return block.

The biggest improvement, compared to my code, would be if the code block may also contain branch and jump instructions. But if the jump targets of these instructions are outside of the code block, it would no longer be possible to compute the stack size of the blocks locally (Section 6.6.3).

6.6.2 Concat block

The concat block is used to concatenate blocks linearly. The concat block of my library is only able to concatenate two blocks. But I think it would be much better if a concat block would be implemented by a list of blocks because in a compiler there are lots of little blocks created and they could be stored in one single list. It would also be good to make it possible to add single instructions to the concat block to avoid the creation of a code block for only one instruction.

6.6.3 Stack computation

The computation of the maximal stack size for a piece of bytecodes program is a complex task in general. By creating a good code block structure it can be simplified. For example in my code block it was not allowed to insert jump or branch instructions and thus a linear analysis is enough to determine the maximal stack size.

One problem of my block structure is that the computation of the stack size is very expensive because it has to be computed several times. To change this, the values should be buffered and only recomputed if something has changed in the code structure.

Another problem is that it is not possible to add jump or branch instructions to the code block. One idea to solve this would be that the stack sizes are no longer computed locally. To compute the maximal stack of a code attribute one can also first generate the whole code and then perform a control flow analysis. This approach is much more complex but it also allows much more flexibility by creating the block structure for the user and the developer.

Appendix A

Library API

A.1 Lower abstraction level

A.1.1 Cluster: root_cluster

Class: CLASS_FILE_COMPONENT

Basic definition for all parts of a Java class file.

```
deferred class
    CLASS_FILE_COMPONENT

feature -- Status report

    is_dump_allowed (file:  JAVA_CLASS_FILE): BOOLEAN
            -- Is the information complete to dump this component to the 'file'?
        ensure
            definition: Result = (file /= void)

feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
            -- Write the content of the component to the 'file'.
        require
            dump_allowed: is_dump_allowed (file)

end
```

Class: CLASS_FILE_TABLE[G->CLASS_FILE_COMPONENT]

Table for the class file components, like they are used in the Java class file.

```
class
    CLASS_FILE_TABLE[G->CLASS_FILE_COMPONENT]

inherit
```

```
    ARRAYED_LIST[G]
    CLASS_FILE_COMPONENT

  create
    make_empty

  feature -- Initialization

    make_empty is
        -- Create an empty class file table.
      ensure
        is_empty: is_empty

  feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
        -- Write the content of the component to the 'file'.

  invariant

    no_void_elements: not has (void)

  end
```

Class: CONSTANT_CLASS_REF

Reference a to constant class in the constant pool

```
  class
    CONSTANT_CLASS_REF

  inherit
    CLASS_FILE_COMPONENT

  feature -- Access

    reference: CONSTANT_CLASS
        -- Reference to the constant pool entry of the interface

  feature -- Status report

    is_dump_allowed (file:  JAVA_CLASS_FILE): BOOLEAN
        -- Is the information complete to dump this component to the 'file'?
      ensure then
        definition: Result = (file /= void and then reference /= void)

  feature -- Element change
```

```
    set_reference (ref:  like reference)
          -- Set 'reference' to 'ref'.
        require
          ref_not_void: ref /= void
        ensure
          reference_assigned: reference = ref

  feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
          -- Write the content of the component to the 'file'.

  end
```

Class: FIELD

Field in the Java class file.

```
  class
    FIELD

  inherit
    CLASS_FILE_COMPONENT

  feature -- Access

    descriptor_index: CONSTANT_UTF8
          -- Reference to the descriptor

    name_index: CONSTANT_UTF8
          -- Reference to the name

    access_flags: FIELD_FLAGS
          -- Access flags of the field

    attributes: ATTRIBUTE_TABLE [FIELD_ATTRIBUTE]
          -- Attributes of the field

  feature -- Status report

    is_dump_allowed (file:  JAVA_CLASS_FILE): BOOLEAN
          -- Is the information complete to dump this component to the 'file'?
        ensure then
          definition: Result = (file /= void and then
            descriptor_index /= void and then
            name_index /= void and then access_flags /= void)
```

```
feature -- Element change

    set_descriptor_index (index:  like descriptor_index)
            -- Set 'descriptor_index' to 'index'.
        require
            index_not_void: index /= void
        ensure
            descriptor_index_assigned: descriptor_index = index

    set_name_index (name:  like name_index)
            -- Set 'name_index' to 'name'.
        require
            name_not_void: name /= void
        ensure
            name_index_assigned: name_index = name

    set_access_flags (flags:  like access_flags)
            -- Set 'access_flags' to 'flags'.
        require
            flags_not_void: flags /= void
        ensure
            access_flags_assigned: access_flags = flags

    set_attributes (attr:  like attributes)
            -- Set 'attributes' to 'attr'.
        ensure
            attributes_assigned: attributes = attr

feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
            -- Write the content of the component to the 'file'.

end
```

Class: JAVA_CLASS

Structure of a whole class in the Java class file.

```
class
    JAVA_CLASS

inherit
    CLASS_FILE_COMPONENT

create
```

```
    make

feature -- Initialization

    make is
            -- Create a java class object.

feature -- Access

    attributes: ATTRIBUTE_TABLE [CLASS_ATTRIBUTE]
            -- Attributes of the class file

    this_name: STRING
            -- Class name
        require
            this_class_not_void: this_class /= void
            name_index_not_void: this_class.name_index /= void
        ensure
            definition: Result = this_class.name_index.value

    super_name: STRING
            -- Name of the super class
        require
            super_class_not_void: super_class /= void
            name_index_not_void: super_class.name_index /= void
        ensure
            definition: Result = super_class.name_index.value

    this_class: CONSTANT_CLASS
            -- Reference to the class name

    super_class: CONSTANT_CLASS
            -- Reference to the super class name

    interfaces: CLASS_FILE_TABLE [CONSTANT_CLASS_REF]
            -- Interfaces of the class

    methods: CLASS_FILE_TABLE [METHOD]
            -- Methods of the class

    fields: CLASS_FILE_TABLE [FIELD]
            -- Fields of the class

    access_flags: CLASS_FLAGS
            -- Access flags of the class

    constant_pool: CONSTANT_POOL [CONSTANT]
```

```
        -- Constant pool of the class

    minor_version: INTEGER
        -- Minor version number of the class file

    major_version: INTEGER
        -- Major version number of the class file

    const: JAVA_CONSTANTS
        -- Reference to the java constants object

feature -- Status report

    is_dump_allowed (file: JAVA_CLASS_FILE): BOOLEAN
        -- Is the information complete to dump this component to the 'file'?
      ensure then
        definition: Result = (file /= void and then
          constant_pool /= void and then
          access_flags /= void and then this_class /= void)

feature -- Element change

    set_constant_pool (pool: like constant_pool)
        -- Set 'constant_pool' to 'pool'.
      require
        pool_not_void: pool /= void
      ensure
        constant_pool_assigned: constant_pool = pool

    set_access_flags (flags: like access_flags)
        -- Set 'access_flags' to 'flags'.
      require
        flags_not_void: flags /= void
      ensure
        access_flags_assigned: access_flags = flags

    set_minor_version (number: like minor_version)
        -- Set 'minor_version' to 'number'.
      require
        valid_number: number >= const.unsigned_2bytes_min and
          number <= const.unsigned_2bytes_max
      ensure
        minor_version_assigned: minor_version = number

    set_major_version (number: like major_version)
        -- Set 'major_version' to 'number'.
      require
```

```
            valid_number: number >= const.unsigned_2bytes_min and
              number <= const.unsigned_2bytes_max
        ensure
            major_version_assigned: major_version = number

    set_this_class (this:  like this_class)
            -- Set 'this_class' to 'this'.
        require
            this_not_void: this /= void
        ensure
            this_class_assigned: this_class = this

    set_super_class (super:  like super_class)
            -- Set 'super_class' to 'super'.
        ensure
            super_class_assigned: super_class = super

    set_interfaces (inter:  like interfaces)
            -- Set 'interfaces' to 'inter'.
        ensure
            interfaces_assigned: interfaces = inter

    set_fields (table:  like fields)
            -- Set 'fields' to 'table'.
        ensure
            fields_assigned: fields = table

    set_methods (table:  like methods)
            -- Set 'methods' to 'table'.
        ensure
            methods_assigned: methods = table

    set_attributes (table:  like attributes)
            -- Set 'attributes' to 'table'.
        ensure
            attributes_assigned: attributes = table

feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
            -- Write the content of the component to the 'file'.

end
```

Class: JAVA_CLASS_FILE

File to write byte by byte. Longer units are written in big-endian order.

```
class
    JAVA_CLASS_FILE

inherit
    RAW_FILE

create
    make_open_write

feature -- Initialization

    make_open_write (fn:  STRING)
            -- Create file object with 'fn' as file name
            -- and open file for writing;
            -- create it if it does not exist.

feature -- Access

    const: JAVA_CONSTANTS
            -- Reference to the java constants object

feature -- Output

    put_byte (byte:  INTEGER_8)
            -- Write 'byte' as signed byte, big-endian, to the file.

    put_short (short:  INTEGER_16)
            -- Write 'short' as signed short, big-endian, to the file.

    put_int (int:  INTEGER)
            -- Write 'int' as signed int, big-endian, to the file.

    put_long (long:  INTEGER_64)
            -- Write 'long' as signed long, big-endian, to the file.

    put_unsigned_byte (byte:  INTEGER)
            -- Write 'byte as unsigned byte, big-endian, to the file.
        require
            valid_value: (byte >= const.unsigned_1byte_min) and
                (byte <= const.unsigned_1byte_max)

    put_unsigned_short (short:  INTEGER)
            -- Write 'short' as unsigned short, big-endian, to the file.
        require
```

```
            valid_value: (short >= const.unsigned_2bytes_min) and
              (short <= const.unsigned_2bytes_max)

        put_float (float:  REAL)
            -- Write 'float' as signed float, big-endian, to the file.

        put_double (double:  DOUBLE)
            -- Write 'double' as signed double, big-endian, to the file.

        put_utf8_string (utf8_string:  STRING)
            -- Write 'utf8_string' as UTF-8 string, big-endian (high byte first),
            -- to the file.
          require
            not_void: utf8_string /= void

    invariant

        const_not_void: const /= void

    end
```

Class: METHOD

Method in the Java class file.

```
    class
        METHOD

    inherit
        CLASS_FILE_COMPONENT

    feature -- Access

        descriptor_index: CONSTANT_UTF8
            -- Reference to the descriptor

        name_index: CONSTANT_UTF8
            -- Reference to the name

        attributes: ATTRIBUTE_TABLE [METHOD_ATTRIBUTE]
            -- Attributes of the method

        access_flags: METHOD_FLAGS
            -- Access flags of the method

    feature -- Status report
```

```
    is_dump_allowed (file: JAVA_CLASS_FILE): BOOLEAN
            -- Is the information complete to dump this component to the 'file'?
        ensure then
            definition: Result = (file /= void and then
                descriptor_index /= void and then
                name_index /= void and then access_flags /= void)

feature -- Element change

    set_descriptor_index (index: like descriptor_index)
            -- Set 'descriptor_index' to 'index'.
        require
            index_not_void: index /= void
        ensure
            descriptor_index_assigned: descriptor_index = index

    set_name_index (name: like name_index)
            -- Set 'name_index' to 'name'.
        require
            name_not_void: name /= void
        ensure
            name_index_assigned: name_index = name

    set_access_flags (flags: like access_flags)
            -- Set 'access_flags' to 'flags'.
        require
            flags_not_void: flags /= void
        ensure
            access_flags_assigned: access_flags = flags

    set_attributes (attr: like attributes)
            -- Set 'attributes' to 'attr'.
        ensure
            attributes_assigned: attributes = attr

feature -- Basic operations

    dump_component (file: JAVA_CLASS_FILE)
            -- Write the content of the component to the 'file'.

end
```

Class: SINGLETON

Object that creates one single instance of JAVA_CONSTANTS

```
class
```

```
    SINGLETON

  feature -- Access

    singleton: JAVA_CONSTANTS
        -- Singleton object
      ensure
        singleton_created: singleton_created
        singleton_not_void: Result /= void

  feature -- Status report

    singleton_created: BOOLEAN
        -- Has singleton already been created?

  end
```

A.1.2 Cluster: access_flags

Class: ACCESS_FLAGS

Access flags in the Java class file.

```
  deferred class
    ACCESS_FLAGS

  inherit
    CLASS_FILE_COMPONENT

  feature -- Initialization

    make is
        -- Create a new access flags object.

  feature -- Access

    access_flags: INTEGER
        -- Value of the access flags

    const: JAVA_CONSTANTS
        -- Reference to the java constants object

  feature -- Status report

    is_flag_set (b:  INTEGER): BOOLEAN
        -- Is flag 'b' set?
      ensure
```

```
            definition: Result = (access_flags & b /= 0)

        valid_flags (flags:  INTEGER): BOOLEAN
            -- Are the 'flags' valid?

    feature -- Element change

        set_access_flags (flags:  like access_flags)
            -- Set 'access_flags' to 'flags'.
        require
            valid_flags: valid_flags (flags)
        ensure
            access_flags_assigned: access_flags = flags

        add_access_flags (flags:  like access_flags)
            -- Add 'flags' to the current flags (or).
        require
            valid_flags: valid_flags (flags)
        ensure
            access_flags_updated: access_flags = old access_flags | flags

    feature -- Basic operations

        dump_component (file:  JAVA_CLASS_FILE)
            -- Write the content of the component to the 'file'.

    invariant

        valid_access_flags: valid_flags (access_flags)
        const_not_void: const /= void

    end
```

Class: CLASS_FLAGS

Access flags for the class in the Java class file.

```
    class
        CLASS_FLAGS

    inherit
        ACCESS_FLAGS

    create
        make

    feature -- Status report
```

```
    valid_flags (flags: INTEGER): BOOLEAN
            -- Are the 'flags' valid?

end
```

Class: FIELD_FLAGS

Access flags for a filed in the Java class file.

```
class
    FIELD_FLAGS

inherit
    ACCESS_FLAGS

create
    make

feature -- Status report

    valid_flags (flags: INTEGER): BOOLEAN
            -- Are the 'flags' valid?

end
```

Class: INNER_CLASS_FLAGS

Access flags for an inner class in the Java class file.

```
class
    INNER_CLASS_FLAGS

inherit
    ACCESS_FLAGS

create
    make

feature -- Status report

    valid_flags (flags: INTEGER): BOOLEAN
            -- Are the 'flags' valid?

end
```

Class: METHOD_FLAGS

Access flags for a method in the Java class file.

```
class
    METHOD_FLAGS

inherit
    ACCESS_FLAGS

create
    make

feature -- Status report

    valid_flags (flags:  INTEGER): BOOLEAN
            -- Are the 'flags' valid?

end
```

A.1.3 Cluster: attributes

Class: ATTRIBUTE

Attributes in a Java class file.

```
deferred class
    ATTRIBUTE

inherit
    CLASS_FILE_COMPONENT

feature -- Access

    name: STRING
            -- Name of the attribute

    name_index: CONSTANT_UTF8
            -- Reference to the name of the attribute

    length: INTEGER
            -- Length of the attribute in bytes
        require
            is_length_allowed: is_length_allowed

feature -- Status report

    is_dump_allowed (file:  JAVA_CLASS_FILE): BOOLEAN
```

```
        -- Is the information complete to dump this component to the 'file'?
    ensure then
        definition: Result = (file /= void and then name_index /= void)

  is_length_allowed: BOOLEAN
        -- Is it valid to call the feature 'length'?

  is_name_index_valid (const: CONSTANT_UTF8): BOOLEAN
        -- Is 'const' a valid name index?
    ensure
        definition: Result = (const /= void and then
            const.value.same_string (name))

feature -- Element change

  set_name_index (const: CONSTANT_UTF8)
        -- Set 'name_index' to 'const'.
    require
        is_name_index_valid: is_name_index_valid (const)
    ensure then
        name_index_assigned: name_index = const

invariant

  valid_name: name_index = void or else
    (name_index.value /= void and then
    name_index.value.same_string (name))

end
```

Class: ATTRIBUTE_TABLE[G->ATTRIBUTE]

Table to store attributes.

```
class
  ATTRIBUTE_TABLE[G->ATTRIBUTE]

inherit
  CLASS_FILE_TABLE[G]

create
  make_empty

feature -- Measurement

  length: INTEGER
        -- Length of the table in bytes
```

invariant

entry_at_least_one_byte: `length >= count`

end

Class: CLASS_ATTRIBUTE

Attributes that can be attached to a class.

deferred class
`CLASS_ATTRIBUTE`

inherit
`ATTRIBUTE`

end

Class: CODE_ATTRIBUTE

Code attribute of the Java class file.

class
`CODE_ATTRIBUTE`

inherit
`METHOD_ATTRIBUTE`

feature -- Access

Name: `STRING is "Code"`
-- Name of the attribute

length: `INTEGER`
-- Length of the attribute in bytes

code: `INSTRUCTION_LIST [INSTRUCTION]`
-- Instructions of a method

exception_table: `CLASS_FILE_TABLE [EXCEPTION_INFO]`
-- Table of exception handlers in the code

attributes: `ATTRIBUTE_TABLE [CODE_ATTRIBUTE_ATTRIBUTE]`
-- Attributes of the code attribute

```
feature -- Status report

    is_dump_allowed (file:  JAVA_CLASS_FILE): BOOLEAN
            -- Is the information complete to dump this component to the 'file'?
        ensure then
            definition: Result = (file /= void and then
                name_index /= void and then code /= void)

    is_length_allowed: BOOLEAN
            -- Is it valid to call the feature 'length'?
        ensure then
            definition: Result = (code /= void)

feature -- Element change

    set_code (c:  like code)
            -- Set 'code' to 'c'.
        require
            code_not_void: c /= void
            code_not_empty: c.code_length > 0
        ensure
            code_assigned: code = c

    set_exception_table (table:  like exception_table)
            -- Set 'exception_table' to 'table'.
        ensure
            exception_table_assigned: exception_table = table

    set_attributes (attr:  like attributes)
            -- Set 'attributes' to 'attr'.
        ensure
            attributes_assigned: attributes = attr

feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
            -- Write the content of the component to the 'file'.

invariant

    code_table_not_emtpy: (code = void) or else (code.code_length > 0)

end
```

Class: CODE_ATTRIBUTE_ATTRIBUTE

Attributes that can be attached to a code attribute.

```
deferred class
    CODE_ATTRIBUTE_ATTRIBUTE

inherit
    ATTRIBUTE

end
```

Class: CONSTANT_VALUE_ATTRIBUTE

Constant value attribute of the Java class file.

```
class
    CONSTANT_VALUE_ATTRIBUTE

inherit
    FIELD_ATTRIBUTE

feature -- Access

    Name: STRING is "ConstantValue"
            -- Name of the attribute

    constant_value_index: VALUE_CONSTANT
            -- Reference to the value

    Length: INTEGER is 2
            -- Length of the attribute in bytes

feature -- Status report

    is_dump_allowed (file:  JAVA_CLASS_FILE): BOOLEAN
            -- Is the information complete to dump this component to the 'file'?
        ensure then
            definition: Result = (file /= void and then
                name_index /= void and then constant_value_index /= void)

    Is_length_allowed: BOOLEAN is True
            -- Is it valid to call the feature 'length'?

feature -- Element change

    set_constant_value_index (const:  like constant_value_index)
            -- Set 'constant_value_index' to 'const'.
```

```
        require
            index_not_void: const /= void
        ensure
            constant_value_index_assigned: constant_value_index = const

    feature -- Basic operations

        dump_component (file:  JAVA_CLASS_FILE)
                -- Write the content of the component to the 'file'.

    invariant

        is_length_allowed: is_length_allowed
        length_is_valid: length = 2

    end
```

Class: DEPRECATED_ATTRIBUTE

Deprecated attribute of the Java class file.

```
    class
        DEPRECATED_ATTRIBUTE

    inherit
        CLASS_ATTRIBUTE
        FIELD_ATTRIBUTE
        METHOD_ATTRIBUTE

    feature -- Access

        Name: STRING is "Deprecated"
                -- Name of the attribute

        Length: INTEGER is 0
                -- Length of the attribute in bytes

    feature -- Status report

        Is_length_allowed: BOOLEAN is True
                -- Is it valid to call the feature 'length'?

    feature -- Basic operations

        dump_component (file:  JAVA_CLASS_FILE)
                -- Write the content of the component to the 'file'.
```

```
invariant

    is_length_allowed: is_length_allowed
    length_is_valid: length = 0

end
```

Class: EXCEPTION_INFO

Content of the exception table in the code attribute.

```
class
    EXCEPTION_INFO

inherit
    CLASS_FILE_COMPONENT

feature -- Access

    start_pc: INSTRUCTION
            -- Start of exception range (first instruction)

    end_pc: INSTRUCTION
            -- End of exception range (last instruction)

    handler_pc: INSTRUCTION
            -- start of exception handler

    catch_type: CONSTANT_CLASS
            -- Exception type

feature -- Status report

    is_dump_allowed (file:  JAVA_CLASS_FILE): BOOLEAN
            -- Is the information complete to dump this component tot the 'file'?
        ensure then
            definition: Result = (file /= void and then
                start_pc /= void and then end_pc /= void and then
                handler_pc /= void and then catch_type /= void and then
                start_pc.address <= end_pc.address)

feature -- Element change

    set_start_pc (pc:  like start_pc)
            -- Set 'start_pc' to 'pc'.
        require
            pc_not_void: pc /= void
```

```
        ensure
            start_pc_assigned: start_pc = pc

        set_end_pc (pc:  like end_pc)
                -- Set 'end_pc' to 'pc'.
            require
                pc_not_void: pc /= void
            ensure
                end_pc_assigned: end_pc = pc

        set_handler_pc (pc:  like handler_pc)
                -- Set 'handler_pc' to 'pc'.
            require
                pc_not_void: pc /= void
            ensure
                handler_pc_assigned: handler_pc = pc

        set_catch_type (type:  like catch_type)
                -- Set 'catch_type' to 'type'.
            require
                type_not_void: type /= void
            ensure
                catch_type_assigned: catch_type = type

    feature -- Basic operations

        dump_component (file:  JAVA_CLASS_FILE)
                -- Write the content of the component to the 'file'.

    end
```

Class: EXCEPTIONS_ATTRIBUTE

Exceptions attribute of the Java class file.

```
    class
        EXCEPTIONS_ATTRIBUTE

    inherit
        METHOD_ATTRIBUTE

    feature -- Access

        Name: STRING is "Exceptions"
                -- Name of the attribute

        length: INTEGER
```

```
            -- Length of the attribute in bytes
        ensure then
            length_definition: Result = exception_index_length *
                exception_index_table.count + count_length

    exception_index_table: CLASS_FILE_TABLE [CONSTANT_CLASS_REF]
            -- exception index table

feature -- Status report

    is_dump_allowed (file:  JAVA_CLASS_FILE): BOOLEAN
            -- Is the information complete to dump this component to the 'file'?
        ensure then
            definition: Result = (file /= void and then
                name_index /= void and then exception_index_table /= void)

    is_length_allowed: BOOLEAN
            -- Is it valid to call the feature 'length'?
        ensure then
            definition: Result = (exception_index_table /= void)

feature -- Element change

    set_exception_index_table (table:  like exception_index_table)
            -- Set 'exception_index_table' to 'table'.
        require
            table_not_void: table /= void
        ensure
            exception_index_table_assigned: exception_index_table = table

feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
            -- Write the content of the component to the 'file'.

end
```

Class: FIELD_ATTRIBUTE

Attribute that can be attached to a filed.

```
deferred class
    FIELD_ATTRIBUTE

inherit
    ATTRIBUTE
```

```
end
```

Class: INNER_CLASS_INFO

Content of the classes table in the inner class attribute.

```
class
    INNER_CLASS_INFO

inherit
    CLASS_FILE_COMPONENT

feature -- Access

    inner_class_info: CONSTANT_CLASS
            -- Reference to the inner class

    outer_class_info: CONSTANT_CLASS
            -- Reference to the class, containing the inner class

    inner_name: CONSTANT_UTF8
            -- Reference the name of the inner class, if present, else void

    inner_class_access_flags: INNER_CLASS_FLAGS
            -- Access flags of the inner class

feature -- Status report

    is_dump_allowed (file:  JAVA_CLASS_FILE): BOOLEAN
            -- Is the information complete to dump this component to the 'file'?
        ensure then
            definition: Result = (file /= void and then
                inner_class_access_flags /= void)

feature -- Element change

    set_inner_class_info (info:  like inner_class_info)
            -- Set 'inner_class_info' to 'info'.
        ensure
            inner_class_info_assigned: inner_class_info = info

    set_outer_class_info (info:  like outer_class_info)
            -- Set 'outer_class_info' to 'info'.
        ensure
            outer_class_info_assigned: outer_class_info = info

    set_inner_name (name:  like inner_name)
```

```
        -- Set 'inner_name' to 'name'.
      ensure
        inner_name_assigned: inner_name = name

    set_inner_class_access_flags (flags:  like inner_class_access_flags)
        -- Set 'inner_class_access_flags' to 'flags'.
      require
        flags_not_void: flags /= void
      ensure
        inner_class_access_flags_assigned:
          inner_class_access_flags = flags

  feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
        -- Write the content of the component to the 'file'.

  end
```

Class: INNER_CLASSES_ATTRIBUTE

Inner classes attribute of the Java class file.

```
  class
    INNER_CLASSES_ATTRIBUTE

  inherit
    CLASS_ATTRIBUTE

  feature -- Access

    Name: STRING is "InnerClasses"
        -- Name of the attribute

    length: INTEGER
        -- Length of the attribute in bytes
      ensure then
        length_definition: Result = class_entry_size *
          classes.count + count_size

    classes: CLASS_FILE_TABLE [INNER_CLASS_INFO]
        -- Table of inner clases

  feature -- Status report

    is_dump_allowed (file:  JAVA_CLASS_FILE): BOOLEAN
        -- Is the information complete to dump this component to the 'file'?
```

```
        ensure then
            definition: Result = (file /= void and then
                name_index /= void and then classes /= void)

    is_length_allowed: BOOLEAN
            -- Is it valid to call the feature 'length'?
        ensure then
            definition: Result = (classes /= void)

feature -- Element change

    set_classes (table:  like classes)
            -- Set 'classes' to 'table'.
        require
            table_not_void: table /= void
        ensure
            classes_assigned: classes = table

feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
            -- Write the content of the component to the 'file'.

end
```

Class: LINE_NUMBER_INFO

The content of the line number table in the line number table attribute.

```
class
    LINE_NUMBER_INFO

inherit
    CLASS_FILE_COMPONENT

create
    make

feature -- Initialization

    make is
            -- Create a line number info object.

feature -- Access

    start_pc: INSTRUCTION
            -- Start of the code area
```

```
    line_number: INTEGER
        -- Line number of the code area

    const: JAVA_CONSTANTS
        -- Reference to the java constants object

feature -- Status report

    is_dump_allowed (file: JAVA_CLASS_FILE): BOOLEAN
        -- Is the information complete to dump this component to the 'file'?
      ensure then
        definition: Result = (file /= void and then start_pc /= void)

    Is_length_allowed: BOOLEAN is True
        -- Is it valid to call the feature 'length'?

feature -- Element change

    set_start_pc (pc: like start_pc)
        -- Set 'start_pc' to 'pc'.
      require
        pc_not_void: pc /= void
      ensure
        start_pc_assigned: start_pc = pc

    set_line_number (number: like line_number)
        -- Set 'line_number' to 'number'.
      require
        valid_line_number: (number >= const.unsigned_2bytes_min) and
          (number <= const.unsigned_2bytes_max)
      ensure
        line_number_assigned: line_number = number

feature -- Basic operations

    dump_component (file: JAVA_CLASS_FILE)
        -- Write the content of the component to the 'file'.

invariant

    line_number_is_valid: (line_number >= const.unsigned_2bytes_min) and
      (line_number <= const.unsigned_2bytes_max)

end
```

Class: LINE_NUMBER_TABLE_ATTRIBUTE

Line number table attribute of the Java class file.

```
class
    LINE_NUMBER_TABLE_ATTRIBUTE

inherit
    CODE_ATTRIBUTE_ATTRIBUTE

feature -- Access

    Name: STRING is "LineNumberTable"
            -- Name of the attribute

    length: INTEGER
            -- Length of the attribute in bytes
        ensure then
            length_definition: Result = table_entry_size *
                line_number_table.count + count_size

    line_number_table: CLASS_FILE_TABLE [LINE_NUMBER_INFO]
            -- Line number table

feature -- Status report

    is_dump_allowed (file:  JAVA_CLASS_FILE): BOOLEAN
            -- Is the information complete to dump this component to the 'file'?
        ensure then
            definition: Result = (file /= void and then
                name_index /= void and then line_number_table /= void)

    is_length_allowed: BOOLEAN
            -- Is is valid to call the feature 'length'?
        ensure then
            definition: Result = (line_number_table /= void)

feature -- Element change

    set_line_number_table (table:  like line_number_table)
            -- Set 'line_number_table' to 'table'.
        require
            table_not_void: table /= void
        ensure
            line_number_table_assigned: line_number_table = table

feature -- Basic operations
```

```
    dump_component (file: JAVA_CLASS_FILE)
            -- Write the content of the component to the 'file'.

end
```

Class: LOCAL_VARIABLE_INFO

Content of the local varible table in the local variable table attribute.

```
class
    LOCAL_VARIABLE_INFO

inherit
    CLASS_FILE_COMPONENT

create
    make

feature -- Initialization

    make is
            -- Create a local variable info object.

feature -- Access

    start_pc: INSTRUCTION
            -- Range start, where local variable has a value

    end_pc: INSTRUCTION
            -- Range length, where local variable has a value

    name: CONSTANT_UTF8
            -- Name of the local variable

    descriptor: CONSTANT_UTF8
            -- Descriptor of the local variable

    index: INTEGER
            -- Index in the array of the current frame

    const: JAVA_CONSTANTS
            -- Reference to the java constants object

feature -- Status report

    is_dump_allowed (file: JAVA_CLASS_FILE): BOOLEAN
            -- Is the information complete to dump this component to the 'file'?
```

```
      ensure then
        definition: Result = (file /= void and then
          start_pc /= void and then end_pc /= void and then
          name /= void and then descriptor /= void and then
          end_pc.address - start_pc.address >= 0)

  feature -- Element change

    set_start_pc (pc:  like start_pc)
        -- Set 'start_pc' to 'pc'.
      require
        pc_not_void: pc /= void
      ensure
        start_pc_assigned: start_pc = pc

    set_end_pc (pc:  like end_pc)
        -- Set 'end_pc' to 'pc'.
      require
        pc_not_void: pc /= void
      ensure
        end_pc_assigned: end_pc = pc

    set_name (n:  like name)
        -- Set 'name' to 'n'.
      require
        n_not_void: n /= void
      ensure
        name_assigned: name = n

    set_descriptor (des:  like descriptor)
        -- Set 'descriptor' to 'des'.
      require
        des_not_void: des /= void
      ensure
        descriptor_assigned: descriptor = des

    set_index (ind:  like index)
        -- Set 'index' to 'ind'.
      require
        valid_index: (ind >= const.unsigned_2bytes_min) and
          (ind <= const.unsigned_2bytes_max)
      ensure
        index_assigned: index = ind

  feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
```

```
        -- Write the content of the component to the 'file'.

    invariant

        index_is_valid: (index >= const.unsigned_2bytes_min) and
            (index <= const.unsigned_2bytes_max)

    end
```

Class: LOCAL_VARIABLE_TABLE_ATTRIBUTE

Local variable table attribute of a Java class file.

```
    class
        LOCAL_VARIABLE_TABLE_ATTRIBUTE

    inherit
        CODE_ATTRIBUTE_ATTRIBUTE

    feature -- Access

        Name: STRING is "LocalVariableTable"
            -- Name of the attribute

        length: INTEGER
            -- Length of the attribute in bytes
          ensure then
            definition: Result = table_entry_size *
                local_variable_table.count + count_size

        local_variable_table: CLASS_FILE_TABLE [LOCAL_VARIABLE_INFO]
            -- Local variable info table

    feature -- Status report

        is_dump_allowed (file:  JAVA_CLASS_FILE): BOOLEAN
            -- Is the information complete to dump this component to the 'file'?
          ensure then
            definition: Result = (file /= void and then
                name_index /= void and then local_variable_table /= void)

        is_length_allowed: BOOLEAN
            -- Is it valid to call the feature 'length'?
          ensure then
            definition: Result = (local_variable_table /= void)

    feature -- Element change
```

```
    set_local_variable_table (table:  like local_variable_table)
            -- Set 'local_variable_table' to 'table'.
        require
            table_not_void: table /= void
        ensure
            local_varibale_table_assigned: local_variable_table = table

feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
            -- Write the content of the component to the 'file'.

end
```

Class: METHOD_ATTRIBUTE

Attributes that can be attached to a method.

```
deferred class
    METHOD_ATTRIBUTE

inherit
    ATTRIBUTE

end
```

Class: SOURCE_FILE_ATTRIBUTE

Source file attribute of the Java class file.

```
class
    SOURCE_FILE_ATTRIBUTE

inherit
    CLASS_ATTRIBUTE

feature -- Access

    Name: STRING is "SourceFile"
            -- Name of the attribute

    source_file_index: CONSTANT_UTF8
            -- Reference to the source file

    Length: INTEGER is 2
```

```
        -- Length of the attribute in bytes

    feature -- Status report

        is_dump_allowed (file: JAVA_CLASS_FILE): BOOLEAN
            -- Is the information complete to dump this component to the 'file'?
        ensure then
            definition: Result = (file /= void and then
                name_index /= void and then source_file_index /= void)

        Is_length_allowed: BOOLEAN is True
            -- Is it valid to call the feature 'length'?

    feature -- Element change

        set_source_file_index (const: like source_file_index)
            -- Set 'source_file_index' to 'const'.
        require
            const_not_void: const /= void
        ensure
            source_file_index_assigned: source_file_index = const

    feature -- Basic operations

        dump_component (file: JAVA_CLASS_FILE)
            -- Write the content of the component to the 'file'.

    invariant

        is_length_allowed: is_length_allowed
        length_is_valid: length = 2

    end
```

Class: SYNTHETIC_ATTRIBUTE

Synthetic attribute of the Java class file.

```
    class
        SYNTHETIC_ATTRIBUTE

    inherit
        FIELD_ATTRIBUTE
        METHOD_ATTRIBUTE

    feature -- Access
```

Name: `STRING is "Synthetic"`
-- Name of the attribute

Length: `INTEGER is 0`
-- Length of the attribute in bytes

feature -- Status report

Is_length_allowed: `BOOLEAN is True`
-- Is it valid to call the feature 'length'?

feature -- Basic operations

dump_component `(file: JAVA_CLASS_FILE)`
-- Write the content of the component to the 'file'.

invariant

is_length_allowed: `is_length_allowed`
length_is_valid: `length = 0`

end

A.1.4 Cluster: constant_pool

Class: CONSTANT

Constant in the constant pool of the Java class file.

deferred class
`CONSTANT`

inherit
`CLASS_FILE_COMPONENT`

feature -- Initialization

make `is`
-- Create a new constant object.

feature -- Access

tag: `INTEGER`
-- Tag of the constant

size: `INTEGER`
-- Size of the constant in the constant pool

```
    index: INTEGER
        -- Index in the constant pool

    const: JAVA_CONSTANTS
        -- Reference to the java constants object

  feature {CONSTANT_POOL} -- Implementation

    set_index (value:  like index)
        -- Set 'index' to 'value'.
      require
        valid_index: value >= 1
      ensure
        index_updated: value = index

  invariant

    const_not_void: const /= void

  end
```

Class: CONSTANT_CLASS

Class constant in the constant pool.

```
  class
    CONSTANT_CLASS

  inherit
    CONSTANT

  create
    make

  feature -- Access

    tag: INTEGER
        -- Tag of the constant
      ensure then
        definition: tag = const.constant_class_tag

    Size: INTEGER is 1
        -- Size of the constant

    name_index: CONSTANT_UTF8
        -- Reference to the name of the class
```

```
feature -- Status report

  is_dump_allowed (file: JAVA_CLASS_FILE): BOOLEAN
      -- Is the information complete to dump this component to the 'file'?
    ensure then
      definition: Result = (file /= void and then name_index /= void)

feature -- Element change

  set_name_index (c: like name_index)
      -- Set 'name_index' to 'c'.
    require
      not_void: c /= void
    ensure
      name_index_assigned: name_index = c

feature -- Basic operations

  dump_component (file: JAVA_CLASS_FILE)
      -- Write the content of the component to the 'file'.

invariant

  size_is_valid: size = 1

end
```

Class: CONSTANT_DOUBLE

Double constant in the constant pool.

```
class
  CONSTANT_DOUBLE

inherit
  VALUE_8BYTES_CONSTANT

create
  make

feature -- Access

  tag: INTEGER
      -- Tag of the constant
    ensure then
      definition: tag = const.constant_double_tag
```

```
    value: DOUBLE
        -- Double value of the constant

  feature -- Element change

    set_value (d:  like value)
        -- Set 'value' to 'd'.
      ensure
        value_assigned: value = d

  feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
        -- Write the content of the component to the 'file'.

  end
```

Class: CONSTANT_FIELD_REF

Field ref constant in the constant pool.

```
  class
    CONSTANT_FIELD_REF

  inherit
    REF_CONSTANT

  create
    make

  feature -- Access

    tag: INTEGER
        -- Tag of the constant
      ensure then
        definition: tag = const.constant_field_ref_tag

    Size: INTEGER is 1
        -- Size of the constant

  invariant

    size_is_valid: size = 1

  end
```

Class: CONSTANT_FLOAT

Float constant in the constant pool.

```
class
    CONSTANT_FLOAT

inherit
    VALUE_4BYTES_CONSTANT

create
    make

feature -- Access

    tag: INTEGER
            -- Tag of the constant
        ensure then
            definition: tag = const.constant_float_tag

    value: REAL
            -- Float value of the constant

feature -- Element change

    set_value (r:  like value)
            -- Set 'value' to 'r'.
        ensure
            value_assigned: value = r

feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
            -- Write the content of the component to the 'file'.

end
```

Class: CONSTANT_INTEGER

Integer constant in the constant pool.

```
class
    CONSTANT_INTEGER

inherit
    VALUE_4BYTES_CONSTANT

create
```

```
    make

  feature -- Access

    tag: INTEGER
        -- Tag of the constant
      ensure then
        definition: tag = const.constant_integer_tag

    value: INTEGER
        -- Integer value of the constant

  feature -- Element change

    set_value (i:  like value)
        -- Set 'value' to 'i'.
      ensure
        value_assigned: value = i

  feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
        -- Write the content of the component to the 'file'.

  end
```

Class: CONSTANT_INTERFACE_METHOD_REF

Interface method ref constant in the constant pool.

```
  class
    CONSTANT_INTERFACE_METHOD_REF

  inherit
    REF_CONSTANT

  create
    make

  feature -- Access

    tag: INTEGER
        -- Tag of the constant
      ensure then
        definition: tag = const.constant_interface_method_ref_tag

    Size: INTEGER is 1
```

```
        -- Size of the constant

    invariant

        size_is_valid: size = 1

    end
```

Class: CONSTANT_LONG

Long constant in the constant pool.

```
    class
        CONSTANT_LONG

    inherit
        VALUE_8BYTES_CONSTANT

    create
        make

    feature -- Access

        tag: INTEGER
                -- Tag of the constant
            ensure then
                definition: tag = const.constant_long_tag

        value: INTEGER_64
                -- Long value of the constant

    feature -- Element change

        set_value (i:  like value)
                -- Set 'value' to 'i'.
            ensure
                value_assigned: value = i

    feature -- Basic operations

        dump_component (file:  JAVA_CLASS_FILE)
                -- Write the content of the component to the 'file'.

    end
```

Class: CONSTANT_METHOD_REF

Method ref constant in the constant pool.

```
class
    CONSTANT_METHOD_REF

inherit
    REF_CONSTANT

create
    make

feature -- Access

    tag: INTEGER
            -- Tag of the constant
        ensure then
            definition: tag = const.constant_method_ref_tag

    Size: INTEGER is 1
            -- Size of the constant

invariant

    size_is_valid: size = 1

end
```

Class: CONSTANT_NAME_AND_TYPE

Name and type constant in the constant pool.

```
class
    CONSTANT_NAME_AND_TYPE

inherit
    CONSTANT

create
    make

feature -- Access

    tag: INTEGER
            -- Tag of the constant
        ensure then
            definition: tag = const.constant_name_and_type_tag
```

```
    Size: INTEGER is 1
          -- Size of the constant

    descriptor_index: CONSTANT_UTF8
          -- Reference to the descriptor

    name_index: CONSTANT_UTF8
          -- Reference to the name

  feature -- Status report

    is_dump_allowed (file:  JAVA_CLASS_FILE): BOOLEAN
          -- Is the information complete to dump this component to the 'file'?
      ensure then
        definition: Result = (file /= void and then
          descriptor_index /= void and then name_index /= void)

  feature -- Element change

    set_descriptor_index (i:  like descriptor_index)
          -- Set 'descriptor_index' to 'i'.
      require
        not_void: i /= void
      ensure
        descriptor_index_assigned: descriptor_index = i

    set_name_index (i:  like name_index)
          -- Set 'name_index' to 'i'.
      require
        not_void: i /= void
      ensure
        name_index_assigned: name_index = i

  feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
          -- Write the content of the component to the 'file'.

  invariant

    size_is_valid: size = 1

  end
```

Class: CONSTANT_POOL[G->CONSTANT]

Constant pool in the Java class file.

```
class
    CONSTANT_POOL[G->CONSTANT]

inherit
    CLASS_FILE_TABLE[G]

create
    make_empty

feature -- Initialization

    make_empty is
            -- Creates empty constant pool.
        ensure then
            constant_pool_count_valid: constant_pool_count = 1

feature -- Access

    constant_pool_count: INTEGER
            -- Size of the constant pool in the class file

feature -- Element change

    force (v:  like item)
            -- Add 'v' as a constant to the constant pool.
        ensure then
            constant_added: count = old count + 1
            cosntant_pool_count_increased: constant_pool_count =
                old constant_pool_count + v.size

    extend (v:  like item)
            -- Add 'v' as a constant to the constant pool.
        ensure then
            constant_added: count = old count + 1
            cosntant_pool_count_increased: constant_pool_count =
                old constant_pool_count + v.size

feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
            -- Write the content of the component to the 'file'.

invariant
```

```
    valid_size: constant_pool_count >= 1

end
```

Class: CONSTANT_STRING

String constant in the constant pool.

```
class
    CONSTANT_STRING

inherit
    VALUE_4BYTES_CONSTANT

create
    make

feature -- Access

    tag: INTEGER
            -- Tag of the constant
        ensure then
            definition: tag = const.constant_string_tag

    string_index: CONSTANT_UTF8
            -- Reference to the content of the string

feature -- Status report

    is_dump_allowed (file:  JAVA_CLASS_FILE): BOOLEAN
            -- Is the information complete to dump this component to the 'file'?
        ensure then
            definition: Result = (file /= void and then
                string_index /= void)

feature -- Element change

    set_string_index (s:  like string_index)
            -- Set 'string_index' to 's'.
        require
            not_void: s /= void
        ensure
            string_index_assigned: string_index = s

feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
```

```
        -- Write the content of the component to the 'file'.

    invariant

        size_is_valid: size = 1

    end
```

Class: CONSTANT_UTF8

UTF8 constant in the constant pool.

```
    class
        CONSTANT_UTF8

    inherit
        CONSTANT

    create
        make

    feature -- Access

        tag: INTEGER
                -- Tag of the constant
            ensure then
                definition: tag = const.constant_utf8_tag

        Size: INTEGER is 1
                -- Size of the constant

        value: STRING
                -- String that the constant contains

        value_unicode: ARRAYED_LIST [INTEGER_16]
                -- String of unicode characters

        length: INTEGER
                -- Number of bytes in the constant
            require
                value_not_void: value /= void
            ensure
                definition: Result = utf8_string.count

    feature -- Status report

        is_dump_allowed (file:  JAVA_CLASS_FILE): BOOLEAN
```

```
            -- Is the information complete to dump this component to the 'file'?
        ensure then
            definition: Result = (file /= void and then value /= void)

    feature -- Element change

        set_value (s:  like value)
            -- Set 'value' to 's'.
        require
            not_void: s /= void
        ensure
            value_assigned: value = s
            utf8_string_not_void: utf8_string /= void

        set_value_unicode (s:  like value_unicode)
            -- Set 'value_unicode' to 's'.
        require
            s_not_void: s /= void
        ensure
            value_unicode_assigned: value_unicode = s
            utf8_string_not_void: utf8_string /= void

    feature -- Basic operations

        dump_component (file:  JAVA_CLASS_FILE)
            -- Write the content of the component to the 'file'.

    invariant

        legal_length: (value = void) or else
            ((length >= const.unsigned_2bytes_min) and
            (length <= const.unsigned_2bytes_max))
        size_is_valid: size = 1

    end
```

Class: REF_CONSTANT

Reference constant to a field, a method or an interface method.

```
    deferred class
        REF_CONSTANT

    inherit
        CONSTANT

    feature -- Access
```

```
    name_and_type_index: CONSTANT_NAME_AND_TYPE
            -- Reference to the name and type information

    class_index: CONSTANT_CLASS
            -- Reference to the class that contains the declaration of the ref

  feature -- Status report

    is_dump_allowed (file:  JAVA_CLASS_FILE): BOOLEAN
            -- Is the information complete to dump this component to the 'file'?
      ensure then
        definition: Result = (file /= void and then
          name_and_type_index /= void and then class_index /= void)

  feature -- Element change

    set_name_and_type_index (i:  like name_and_type_index)
            -- Set 'name_and_type_index' to 'i'.
      require
        not_void: i /= void
      ensure
        name_and_type_index_assigned: name_and_type_index = i

    set_class_index (i:  like class_index)
            -- Set 'class_index' to 'i'.
      require
        not_void: i /= void
      ensure
        class_index_assigned: class_index = i

  feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
            -- Write the content of the component to the 'file'.

  end
```

Class: VALUE_4BYTES_CONSTANT

Constants of the constant pool accessed by a 4 bytes value

```
  deferred class
    VALUE_4BYTES_CONSTANT

  inherit
    VALUE_CONSTANT
```

```
feature -- Access

    Size: INTEGER is 1
            -- Size of the constant

invariant

    size_is_valid: size = 1

end
```

Class: VALUE_8BYTES_CONSTANT

Constants of the constant pool accessed by an 8 bytes value

```
deferred class
    VALUE_8BYTES_CONSTANT

inherit
    VALUE_CONSTANT

feature -- Access

    Size: INTEGER is 2
            -- Size of the constant

invariant

    size_is_valid: size = 2

end
```

Class: VALUE_CONSTANT

Constants of the constant pool with a value

```
deferred class
    VALUE_CONSTANT

inherit
    CONSTANT

end
```

A.1.5 Cluster: instructions

Class: INSTRUCTION

JVM instructions in the Java class files.

```
class
    INSTRUCTION

inherit
    CLASS_FILE_COMPONENT

create
    make

feature -- Initialization

    make (i:  INTEGER)
            -- Create a instruction by giving an opcode 'i'.
        require
            valid_opcode: i >= 0 and i <= 201 and i /= 186
        ensure
            opcode_set: opcode = i

feature -- Access

    opcode: INTEGER
            -- Code of the instruction

    address: INTEGER
            -- Address in the instruction list

    byte_const: INTEGER_8
            -- Signed byte (bipush, iinc)

    short_const: INTEGER_16
            -- Signed short (sipush, wide)

    constant_4bytes: VALUE_4BYTES_CONSTANT
            -- Constant pool reference (ldc, ldc_w)

    constant_8bytes: VALUE_8BYTES_CONSTANT
            -- Constant pool reference (ldc2_w)

    local_variable: INTEGER
            -- Index of a local variable (i, l, f, d, a - load, -store, iinc, ret)

    branch: INSTRUCTION
```

-- Branch target (if<cond>, if_icmp<cond>, goto, jsr,
-- ifnull, ifnonnull, goto_w, jsr_w)

constant_field: `CONSTANT_FIELD_REF`
-- Constant pool reference (getstatic, putstatic, getfield, putfield)

constant_method: `CONSTANT_METHOD_REF`
-- Constant pool reference (invokevirtual, invokespecial, invokestatic)

constant_interface_method: `CONSTANT_INTERFACE_METHOD_REF`
-- Constant pool reference (invokeinterface)

count: `INTEGER`
-- Count value (invokeinterface)

dimensions: `INTEGER`
-- Count value (multianewarray)

constant_class: `CONSTANT_CLASS`
-- Constant pool reference
-- (new, anewarray, checkcast, instanceof, multianewarray)

array_type: `INTEGER`
-- Type of an array (newarray)

default_jump: `INSTRUCTION`
-- Default jump (tableswitch, lookupswitch)

low_switch: `INTEGER`
-- Start of the table (tableswitch)

high_switch: `INTEGER`
-- End of the table (tableswitch)

npairs_switch: `INTEGER`
-- Number of lookup entries (lookupswitch)

jump_offsets: `ARRAY [INSTRUCTION]`
-- Jump offsets in table (tableswitch)

jump_match_offsets: `ARRAY [MATCH_OFFSET]`
-- Jump offsets and matches in table (lookupswitch)

wide_opcode: `INTEGER`
-- Opcode in the wide instruction (wide)

wide_index: `INTEGER`

```
        -- Index of a local variable (wide)

    const: JAVA_CONSTANTS
        -- Reference to the java constants

feature -- Measurement

    length: INTEGER
        -- Length of the instruction in bytes
      require
        wide_opcode_set: (opcode /= const.wide) or else
          (wide_opcode > 0)

    local_variable_use: INTEGER
        -- Maximal local variable of the instruction

    stack_change: INTEGER
        -- How the instruction changes the stack size

feature -- Status report

    is_byte_const_allowed: BOOLEAN
        -- Does 'opcode' accepts byte?
      ensure
        definition: Result = (opcode = const.bipush or
          opcode = const.iinc)

    is_short_const_allowed: BOOLEAN
        -- Does 'opcode' accepts short?
      ensure
        definition: Result = (opcode = const.sipush or
          opcode = const.wide)

    is_constant_4bytes_allowed: BOOLEAN
        -- Does 'opcode' accepts 4 byte constant?
      ensure
        definition: Result = (opcode = const.ldc or
          opcode = const.ldc_w)

    is_constant_8bytes_allowed: BOOLEAN
        -- Does 'opcode' accepts 8 byte constant?
      ensure
        definition: Result = (opcode = const.ldc2_w)

    is_local_variable_allowed: BOOLEAN
        -- Does 'opcode' accepts local variable?
      ensure
```

```
        definition: Result = (((opcode >= const.iload) and
          (opcode <= const.aload)) or ((opcode >= const.istore) and
          (opcode <= const.astore)) or (opcode = const.iinc) or
          (opcode = const.ret))

    is_branch_allowed: BOOLEAN
        -- Does 'opcode' accepts branch target?
      ensure
        definition: Result = (((opcode >= const.ifeq) and
          (opcode <= const.jsr)) or ((opcode >= const.ifnull) and
          (opcode <= const.jsr_w)))

    is_constant_field_allowed: BOOLEAN
        -- Does 'opcode' accepts constant field?
      ensure
        definition: Result = (opcode >= const.getstatic and
          opcode <= const.putfield)

    is_constant_method_allowed: BOOLEAN
        -- Does 'opcode' accepts constant method?
      ensure
        definition: Result = (opcode >= const.invokevirtual and
          opcode <= const.invokestatic)

    is_constant_interface_method_allowed: BOOLEAN
        -- Does 'opcode' accepts constant interface method?
      ensure
        definition: Result = (opcode = const.invokeinterface)

    is_count_allowed: BOOLEAN
        -- Does 'opcode' accepts count value?
      ensure
        definition: Result = (opcode = const.invokeinterface)

    is_dimensions_allowed: BOOLEAN
        -- Does 'opcode' accepts dimensions value?
      ensure
        definition: Result = (opcode = const.multianewarray)

    is_constant_class_allowed: BOOLEAN
        -- Does 'opcode' accepts constant class?
      ensure
        definition: Result = ((opcode = const.new) or
          (opcode = const.anewarray) or
          (opcode = const.checkcast) or
          (opcode = const.instanceof) or
          (opcode = const.multianewarray))
```

```
is_array_type_allowed: BOOLEAN
      -- Does 'opcode' accepts array type?
   ensure
      definition: Result = (opcode = const.newarray)

is_default_jump_allowed: BOOLEAN
      -- Does 'opcode' accepts default jump?
   ensure
      definition: Result = (opcode = const.tableswitch or
         opcode = const.lookupswitch)

is_low_switch_allowed: BOOLEAN
      -- Does 'opcode' accepts low switch value?
   ensure
      definition: Result = (opcode = const.tableswitch)

is_high_switch_allowed: BOOLEAN
      -- Does 'opcode' accepts high switch value?
   ensure
      definition: Result = (opcode = const.tableswitch)

is_npairs_switch_allowed: BOOLEAN
      -- Does 'opcode' accepts npairs switch value?
   ensure
      definition: Result = (opcode = const.lookupswitch)

is_jump_offsets_allowed: BOOLEAN
      -- Does 'opcode' accepts jump offsets?
   ensure
      definition: Result = (opcode = const.tableswitch)

is_jump_match_offsets_allowed: BOOLEAN
      -- Does 'opcode' accepts jump match offsets?
   ensure
      definition: Result = (opcode = const.lookupswitch)

is_wide_opcode_allowed: BOOLEAN
      -- Does 'opcode' accepts wide opcode?
   ensure
      definition: Result = (opcode = const.wide)

is_wide_index_allowed: BOOLEAN
      -- Does 'opcode' accepts wide index?
   ensure
      definition: Result = (opcode = const.wide)
```

```
is_dump_allowed (file: JAVA_CLASS_FILE): BOOLEAN
        -- Is the information complete to dump this component to the 'file'?
    ensure then
        definition: Result = (file /= void and then
            (not is_constant_4bytes_allowed or else
            constant_4bytes /= void) and then
            (not is_constant_8bytes_allowed or else
            constant_8bytes /= void) and then
            (not is_branch_allowed or else branch /= void) and then
            (not is_constant_field_allowed or else
            constant_field /= void) and then
            (not is_constant_method_allowed or else
            constant_method /= void) and then
            (not is_constant_interface_method_allowed or else
            constant_interface_method /= void) and then
            (not is_constant_class_allowed or else
            constant_class /= void) and then
            (not is_default_jump_allowed or else
            default_jump /= void) and then
            (not is_jump_offsets_allowed or else
            jump_offsets /= void) and then
            (not is_jump_match_offsets_allowed or else
            jump_match_offsets /= void) and then
            (opcode /= 18 or else constant_4bytes.index < 256))

is_2_bytes_instruction: BOOLEAN
        -- Does the instruction use two bytes in the class file?

is_3_bytes_instruction: BOOLEAN
        -- Does the instruction use three bytes in the class file?

is_4_bytes_instruction: BOOLEAN
        -- Does the instruction use four bytes in the class file?

is_5_bytes_instruction: BOOLEAN
        -- Does the instruction use five bytes in the class file?

is_local_variable_1_byte: BOOLEAN
        -- Does the instruciton use the local variable specified in the attribute
        -- local variable?

is_local_variable_2_bytes: BOOLEAN
        -- Does the instruction use the local varable specified in the attribute
        -- local variable and the next one?

is_1_local_variable: BOOLEAN
        -- Does the instruction use the first local varaible?
```

```
        is_2_local_variable: BOOLEAN
                -- Does the instruction use the second local varaible?

        is_3_local_variable: BOOLEAN
                -- Does the instruction use the third local varaible?

        is_4_local_variable: BOOLEAN
                -- Does the instruction use the fourth local varaible?

        is_5_local_variable: BOOLEAN
                -- Does the instruction use the fifth local varaible?

        is_stack_consume_4: BOOLEAN
                -- Does the instruction consume 4 stack entries?

        is_stack_consume_3: BOOLEAN
                -- Does the instruction consume 3 stack entries?

        is_stack_consume_2: BOOLEAN
                -- Does the instruction consume 2 stack entries?

        is_stack_consume_1: BOOLEAN
                -- Does the instruction consume 1 stack entry?

        is_stack_produce_1: BOOLEAN
                -- Does the instruction produce 1 stack entry?

        is_stack_produce_2: BOOLEAN
                -- Does the instruction produce 2 stack entries?

    feature -- Element change

        set_byte_const (a_byte:  like byte_const)
                -- Set 'byte_const' to 'a_byte'.
            require
                is_byte_const_allowed: is_byte_const_allowed
            ensure
                byte_const_assigned: byte_const = a_byte

        set_short_const (a_short:  like short_const)
                -- Set 'short_const' to 'a_short'.
            require
                is_short_const_allowed: is_short_const_allowed
            ensure
                short_const_assigned: short_const = a_short
```

```
set_constant_4bytes (a_constant:  like constant_4bytes)
      -- Set 'constant_4bytes' to 'a_constant'.
   require
      not_void: a_constant /= void
      is_constant_4bytes_allowed: is_constant_4bytes_allowed
   ensure
      constant_assigned: constant_4bytes = a_constant

set_constant_8bytes (a_constant:  like constant_8bytes)
      -- Set 'constant_8bytes' to 'a_constant'.
   require
      not_void: a_constant /= void
      is_constant_8bytes_allowed: is_constant_8bytes_allowed
   ensure
      constant_assigned: constant_8bytes = a_constant

set_local_variable (var:  like local_variable)
      -- Set 'local_variable' to 'var'.
   require
      valid_index: (var >= const.unsigned_1byte_min) and
         (var <= const.unsigned_1byte_max)
      is_local_variable_allowed: is_local_variable_allowed
   ensure
      local_variable_assigned: local_variable = var

set_branch (a_branch:  like branch)
      -- Set 'branch' to 'a_branch'.
   require
      not_void: a_branch /= void
      is_branch_allowed: is_branch_allowed
   ensure
      branch_assigned: branch = a_branch

set_constant_field (field:  like constant_field)
      -- Set 'constant_field' to 'field'.
   require
      not_void: field /= void
      is_constant_field_allowed: is_constant_field_allowed
   ensure
      constant_field_assigned: constant_field = field

set_constant_method (method:  like constant_method)
      -- Set 'constant_method' to 'method'.
   require
      not_void: method /= void
      is_conatant_method_allowed: is_constant_method_allowed
   ensure
```

```
      constant_method_assigned: constant_method = method

  set_constant_interface_method
    (method:  like constant_interface_method)
      -- Set 'constant_interface_method' to 'method'.
    require
      not_void: method /= void
      is_constant_interface_method_allowed:
        is_constant_interface_method_allowed
    ensure
      constant_interface_method_assigned:
        constant_interface_method = method

  set_count (a_count:  like count)
      -- Set 'count' to 'a_count'.
    require
      valid_value: (a_count > const.unsigned_1byte_min) and
        (a_count <= const.unsigned_1byte_max)
      is_count_allowed: is_count_allowed
    ensure
      count_assigned: count = a_count

  set_dimensions (a_dimensions:  like dimensions)
      -- Set 'dimensions' to 'a_dimensions'.
    require
      valid_value: (a_dimensions > const.unsigned_1byte_min) and
        (a_dimensions < const.unsigned_1byte_max)
      is_dimensions_allowed: is_dimensions_allowed
    ensure
      dimensions_assigned: dimensions = a_dimensions

  set_constant_class (a_class:  like constant_class)
      -- Set 'constant_class' to 'a_class'.
    require
      not_void: a_class /= void
      is_constant_class_allowed: is_constant_class_allowed
    ensure
      constant_class_assigned: constant_class = a_class

  set_array_type (type:  like array_type)
      -- Set 'array_type' to 'type'.
    require
      valid_type: (type >= 4) and (type <= 11)
      is_array_type_allowed: is_array_type_allowed
    ensure
      array_type_assigned: array_type = type
```

```
set_default_jump (jump:  like default_jump)
        -- Set 'default_jump' to 'jump'.
    require
        not_void: jump /= void
        is_default_jump_allowed: is_default_jump_allowed
    ensure
        default_jump_assigned: default_jump = jump

set_low_switch (low:  like low_switch)
        -- Set 'low_switch' to 'low'.
    require
        is_low_switch_allowed: is_low_switch_allowed
    ensure
        low_switch_assigned: low_switch = low

set_high_switch (high:  like high_switch)
        -- Set 'high_switch' to 'high'.
    require
        is_high_switch_allowed: is_high_switch_allowed
    ensure
        high_switch_assigned: high_switch = high

set_npairs_switch (npairs:  like npairs_switch)
        -- Set 'npairs_switch' to 'npairs'.
    require
        is_npairs_switch_allowed: is_npairs_switch_allowed
    ensure
        npairs_switch_assigned: npairs_switch = npairs

set_jump_offsets (offsets:  like jump_offsets)
        -- Set 'jump_offsets' to 'offsets'.
    require
        not_void: offsets /= void
        no_void_offsets: not offsets.has (void)
    ensure
        jump_offsets_assigned: jump_offsets = offsets

set_jump_match_offsets (match_offset:  like jump_match_offsets)
        -- Set 'jump_match_offsets' to 'match_offset'.
    require
        not_void: match_offset /= void
        no_void_offsets: not match_offset.has (void)
    ensure
        jump_match_offsets_assigned: jump_match_offsets = match_offset

set_wide_opcode (a_opcode:  like wide_opcode)
        -- Set 'wide_opcode' to 'a_opcode'.
```

```
      require
        valid_opcode: ((a_opcode >= const.iload) and
          (a_opcode <= const.aload)) or
          ((a_opcode >= const.istore) and
          (a_opcode <= const.astore)) or a_opcode = const.iinc or
          a_opcode = const.ret
        is_wide_opcode_allowed: is_wide_opcode_allowed
      ensure
        wide_opcode_assigned: wide_opcode = a_opcode

    set_wide_index (a_index:  like wide_index)
        -- Set 'wide_index' to 'a_index'.
      require
        valid_value: (a_index >= 0) and (a_index < 65536)
        is_wide_index_allowed: is_wide_index_allowed
      ensure
        wide_index_assigned: wide_index = a_index

  feature -- Basic operations

    dump_component (file:  JAVA_CLASS_FILE)
        -- Write the content of the component to the 'file'.

  feature {INSTRUCTION_LIST} -- Implementation

    set_address (addr:  INTEGER)
        -- Set 'address' to 'addr'.
      require
        valid_address: (addr >= const.unsigned_2bytes_min) and
          (addr < const.unsigned_2bytes_max)
      ensure
        address_assigned: address = addr

  invariant

    const_not_void: const /= void
    valid_opcode: opcode >= const.min_opcode and
      opcode <= const.max_opcode and opcode /= const.illegal_opcode
    no_void_jump_offsets: jump_offsets /= void implies
      not jump_offsets.has (void)
    no_void_jump_match_offsets: jump_match_offsets /= void implies
      not jump_match_offsets.has (void)
    local_variable_is_valid: (local_variable >= const.unsigned_1byte_min and
      local_variable <= const.unsigned_1byte_max)

  end
```

Class: INSTRUCTION_LIST[G->INSTRUCTION]

Manages the instructions of a method.

```
class
    INSTRUCTION_LIST[G->INSTRUCTION]

inherit
    LINKED_LIST[G]
    CLASS_FILE_COMPONENT

create
    make

feature -- Access

    code_length: INTEGER
            -- Length of the code in bytes

    max_stack: INTEGER
            -- Maximal depth of method stack

    max_locals: INTEGER
            -- Number of local variables

feature -- Element change

    set_max_stack (stack:  like max_stack)
            -- Set 'max_stack' to 'stack'.
        require
            stack_positive: stack >= 0
        ensure
            max_stack_assigned: max_stack = stack

    set_max_locals (loc:  like max_locals)
            -- Set 'max_locals' to 'loc'.
        require
            local_positive: loc >= 0
        ensure
            max_locals_assigned: max_locals = loc

    append (l:  LINKED_LIST [G])
            -- Append a copy of 'l'.
        ensure then
            code_length_increased:
                code_length >= old code_length + l.count

    extend (v:  like item)
```

```
        -- Add 'v' to the end.
      ensure then
        code_length_increased:
          code_length = old code_length + v.length

    force (v:  like item)
        -- Add 'v' to the end.
      ensure then
        code_length_increased:
          code_length = old code_length + v.length

    merge_left (other:  like Current)
        -- Merge 'other' into current structure before cursor
        -- position. Do not move cursor. Empty 'other'.

    merge_right (other:  like Current)
        -- Merge 'other' into current structure after cursor
        -- position. Do not move cursor. Empty 'other'.

    put (v:  like item)
        -- Replace current item with 'v'.

    put_front (v:  like item)
        -- Add 'v' to the beginning.

    put_i_th (v:  like item; i:  INTEGER)
        -- Replace the i-th instruction with 'v'.

    put_left (v:  like item)
        -- Add 'v' before the current position.

    put_right (v:  like item)
        -- Add 'v' after the current position.

    replace (v:  like item)
        -- Replace current item with 'v'.

  feature -- Removal

    prune (v:  like item)
        -- Remove first occurrence of 'v', if any,
        -- after cursor position.
        -- If found, move cursor to right neighbor;
        -- if not, make structure 'exhausted'.

    prune_all (v:  like item)
        -- Remove all occurrences of 'v'.
```

```
            -- (Reference or object equality,
            -- based on 'object_comparison'.)
            -- Leave structure 'exhausted'.

        remove is
            -- Remove current item.
            -- Move cursor to right neighbor
            -- (or 'after' if no right neighbor).

        remove_left is
            -- Remove item to the left of cursor position.
            -- Do not move cursor.

        remove_right is
            -- Remove item to the right of cursor position.
            -- Do not move cursor.

    feature -- Transformation

        swap (i: INTEGER)
            -- Exchange item at 'i'-th position with item
            -- at cursor position.

    feature -- Basic operations

        dump_component (file: JAVA_CLASS_FILE)
            -- Write the content of the component to the 'file'.

    invariant

        instruction_at_least_one_byte: code_length >= count
        max_stack_is_positive: max_stack >= 0
        max_locals_is_positive: max_locals >= 0

    end
```

Class: MATCH_OFFSET

Match-offset pairs in the lookupswitch instruction.

```
    class
        MATCH_OFFSET

    inherit
        CLASS_FILE_COMPONENT

    create
```

```
    make

feature -- Initialization

    make is
            -- Create a new match offset object.
        ensure
            const_not_void: const /= void

feature -- Access

    match: INTEGER
            -- Value that has to be matched

    offset: INSTRUCTION
            -- Target, if value matches

    address: INTEGER
            -- Address of the lookupswitch instruction

    const: JAVA_CONSTANTS
            -- Reference to the java constants object

feature -- Status report

    is_dump_allowed (file: JAVA_CLASS_FILE): BOOLEAN
            -- Is the information complete to dump this component to the 'file'?
        ensure then
            definition: Result = (file /= void and then offset /= void)

feature -- Element change

    set_match (m: like match)
            -- Set 'match' to 'm'.
        ensure
            match_assigned: match = m

    set_offset (instr: like offset)
            -- Set 'offset' to 'instr'.
        require
            instr_not_void: instr /= void
        ensure
            offset_assigned: offset = instr

    set_address (addr: INTEGER)
            -- Set 'address' to 'addr'.
        require
```

```
        valid_address: (addr >= const.unsigned_2bytes_min) and
          (addr < const.unsigned_2bytes_max)
    ensure
      address_assigned: address = addr

feature -- Basic operations

  dump_component (file:  JAVA_CLASS_FILE)
      -- Write the content of the component to the 'file'.

invariant

  const_not_void: const /= void

end
```

A.2 Higher abstraction level

A.2.1 Cluster: generators

Class: ATTRIBUTE_GENERATOR

Generate attributes that already have a name_index.

```
class
    ATTRIBUTE_GENERATOR

create
    make_from_constant_pool,
    make_from_constant_pool_gen

feature -- Initialization

    make_from_constant_pool (pool: CONSTANT_POOL [CONSTANT])
            -- Create a new attribute generator with a constant pool generator
            -- that uses the constant pool 'pool'.
        require
            pool_not_void: pool /= void
        ensure
            pool_assigned: constant_pool_gen.constant_pool = pool

    make_from_constant_pool_gen (gen: CONSTANT_POOL_GENERATOR)
            -- Create a new attribute generator that uses the constant pool
            -- generator 'gen'.
        require
            gen_not_void: gen /= void
        ensure
            gen_assigned: constant_pool_gen = gen

feature -- Access

    constant_pool_gen: CONSTANT_POOL_GENERATOR
            -- Reference to the used constant pool generator

    constant_value_attribute: CONSTANT_VALUE_ATTRIBUTE
            -- New constant value attribute with already set name_index
        ensure
            result_not_void: Result /= void
            name_index_not_void: Result.name_index /= void

    code_attribute: CODE_ATTRIBUTE
            -- New code attribute with already set name_index
        ensure
            result_not_void: Result /= void
```

```
            name_index_not_void: Result.name_index /= void

        exceptions_attribute: EXCEPTIONS_ATTRIBUTE
            -- New exceptions attribute with already set name_index
        ensure
            result_not_void: Result /= void
            name_index_not_void: Result.name_index /= void

        inner_classes_attribute: INNER_CLASSES_ATTRIBUTE
            -- New inner classes attribute with already set name_index
        ensure
            result_not_void: Result /= void
            name_index_not_void: Result.name_index /= void

        synthetic_attribute: SYNTHETIC_ATTRIBUTE
            -- New synthetic attribute with already set name_index
        ensure
            result_not_void: Result /= void
            name_index_not_void: Result.name_index /= void

        source_file_attribute: SOURCE_FILE_ATTRIBUTE
            -- New source file attribute with already set name_index
        ensure
            result_not_void: Result /= void
            name_index_not_void: Result.name_index /= void

        line_number_table_attribute: LINE_NUMBER_TABLE_ATTRIBUTE
            -- New line number table attribute with already set name_index
        ensure
            result_not_void: Result /= void
            name_index_not_void: Result.name_index /= void

        local_variable_table_attribute: LOCAL_VARIABLE_TABLE_ATTRIBUTE
            -- New local variable table attribute with already set name_index
        ensure
            result_not_void: Result /= void
            name_index_not_void: Result.name_index /= void

        deprecated_attribute: DEPRECATED_ATTRIBUTE
            -- New deprecated attribute with already set name_index
        ensure
            result_not_void: Result /= void
            name_index_not_void: Result.name_index /= void

    invariant

        constant_pool_gen_not_void: constant_pool_gen /= void
```

```
end
```

Class: CLASS_GENERATOR

Creates a Java class and sets its content.

```
class
    CLASS_GENERATOR

create
    make,
    make_from_constant_pool

feature -- Initialization

    make (name:  STRING)
            -- Create class generator object that generate a class with the 'name'.
        require
            name_not_void: name /= void
            name_not_emtpy: not name.is_empty
        ensure
            cp_generator_for_class:
                constant_pool_gen.constant_pool = java_class.constant_pool
            name_assigned: java_class.this_name.is_equal (name)

    make_from_constant_pool
        (name:  STRING; pool:  CONSTANT_POOL [CONSTANT])
            -- Create class generator object that generate a class with the 'name'.
            -- and the constant pool 'pool'.
        require
            name_not_void: name /= void
            name_not_empty: not name.is_empty
        ensure
            cp_generator_for_class:
                constant_pool_gen.constant_pool = java_class.constant_pool
            name_assigned: java_class.this_name.is_equal (name)

feature -- Access

    java_class: JAVA_CLASS
            -- Generated Java class

    constant_pool_gen: CONSTANT_POOL_GENERATOR
            -- Constant pool generator used for the class

    attribute_gen: ATTRIBUTE_GENERATOR
```

```
        -- Attribute generator used for the class

    const: JAVA_CONSTANTS
        -- Reference to the java constants object

feature -- Element change

    set_major_version (number: INTEGER)
        -- Set the major version number of the class to 'number'.
      require
        valid_number: number >= const.unsigned_2bytes_min and
          number <= const.unsigned_2bytes_max
      ensure
        major_version_assigned: java_class.major_version = number

    set_minor_version (number: INTEGER)
        -- Set the minor version number of the class to 'number'.
      require
        valid_number: number >= const.unsigned_2bytes_min and
          number <= const.unsigned_2bytes_max
      ensure
        minor_version_assigned: java_class.minor_version = number

    set_super_class (name: STRING)
        -- Set 'name' to the class name as super class of the current class.
      require
        name_not_void: name /= void
        name_not_empty: not name.is_empty
      ensure
        super_class_not_void: java_class.super_class /= void
        super_class_assigned:
          java_class.super_class.name_index.value.is_equal (name)

    set_access_flags (flags: INTEGER)
        -- Set 'flags' to the access falgs of the class.
      require
        valid_flags: java_class.access_flags.valid_flags (flags)
      ensure
        access_flags_assigned:
          java_class.access_flags.access_flags = flags

    add_interface (inter: STRING)
        -- Add the interface 'inter' to the class.
      require
        inter_not_void: inter /= void
        inter_not_empty: not inter.is_empty
      ensure
```

```
            interfaces_not_void: java_class.interfaces /= void

        add_field (field:  FIELD)
            -- Add the field 'field' to the class.
        require
            field_not_void: field /= void
        ensure
            fields_not_void: java_class.fields /= void
            field_assigned: java_class.fields.has (field)

        add_method (method:  METHOD)
            -- Add the method 'method' to the class.
        require
            method_not_void: method /= void
        ensure
            methods_not_void: java_class.methods /= void
            method_assigned: java_class.methods.has (method)

        add_attribute (attr:  CLASS_ATTRIBUTE)
            -- Add the attribute 'attr' to the class.
        require
            attr_not_void: attr /= void
        ensure
            attributes_not_void: java_class.attributes /= void
            attr_added: java_class.attributes.has (attr)

        add_empty_constructor (flags:  INTEGER)
            -- Add an empty constructor with the access flags 'flags'.
            -- The added constructor only calls super().
        require
            valid_flags: const.valid_method_flags (flags)
            super_class_not_void: java_class.super_class /= void
        ensure
            methods_not_void: java_class.methods /= void

    invariant

        java_class_not_void: java_class /= void
        constant_pool_not_void: java_class.constant_pool /= void
        constant_pool_gen_not_void: constant_pool_gen /= void
        class_flags_not_void: java_class.access_flags /= void
        this_class_not_void: java_class.this_class /= void
        const_not_void: const /= void
        attribute_gen_not_void: attribute_gen /= void

    end
```

Class: CONSTANT_POOL_GENERATOR

Creates and manages a constant pool.

```
class
  CONSTANT_POOL_GENERATOR

create
  make,
  make_from_constant_pool

feature -- Initialization

  make is
      -- Create constant pool generator object.

  make_from_constant_pool (pool: CONSTANT_POOL [CONSTANT])
      -- Create constant pool generator object that manages
      -- the constant pool 'pool'.
    require
      pool_not_void: pool /= void

feature -- Access

  constant_class (name: STRING): CONSTANT_CLASS
      -- Reference to a class constant with the given 'name'.
      -- Create class und UTF-8 constants if they do not exist.
    require
      name_not_void: name /= void
      name_not_empty: not name.is_empty
    ensure
      class_in_pool: has_constant_class (name)
      result_not_void: Result /= void

  constant_field_ref
    (class_name, name, descriptor: STRING): CONSTANT_FIELD_REF
      -- Reference to a field ref constant of the class 'class_name',
      -- with the field name 'name' and the 'descriptor'. Create field ref,
      -- class, name and type and UTF-8 constants if they do not exist.
    require
      class_name_not_void: class_name /= void
      name_not_void: name /= void
    ensure
      field_ref_in_pool:
        has_constant_field_ref (class_name, name, descriptor)
      result_not_void: Result /= void

  constant_method_ref
```

```
    (class_name, name, descriptor:  STRING): CONSTANT_METHOD_REF
      -- Reference to a method ref constant of the class 'class_name',
      -- with the method name 'name' and the 'descriptor'. Create method ref,
      -- class, name and type and UTF-8 constants if they do not exist.
    require
      class_name_not_void: class_name /= void
      name_not_void: name /= void
    ensure
      method_ref_in_pool:
        has_constant_method_ref (class_name, name, descriptor)
      result_not_void: Result /= void

  constant_interface_method_ref
    (class_name, name, descriptor:  STRING):
        CONSTANT_INTERFACE_METHOD_REF
      -- Reference to a interface method ref constant of the class 'class_name',
      -- with the interface method name 'name' and the 'descriptor'.
      -- Create interface method ref, class, name and type and UTF-8 constants
      -- if they do not exist.
    require
      class_name_not_void: class_name /= void
      name_not_void: name /= void
    ensure
      interface_method_ref_in_pool: has_constant_interface_method_ref
        (class_name, name, descriptor)
      result_not_void: Result /= void

  constant_string (string:  STRING): CONSTANT_STRING
      -- Reference to a string constant with the given 'string'.
      -- Create string and UTF-8 constant if they do not exist.
    require
      string_not_void: string /= void
      string_not_empty: not string.is_empty
    ensure
      string_in_pool: has_constant_string (string)
      result_not_void: Result /= void

  constant_string_unicode
    (string:  ARRAYED_LIST [INTEGER_16]):  CONSTANT_STRING
      -- Reference to a string constant with the given 'string'.
      -- Create string and UTF-8 constant if they do not exist.
    require
      string_not_void: string /= void
      string_not_empty: not string.is_empty
    ensure
      string_in_pool: has_constant_string_unicode (string)
      result_not_void: Result /= void
```

```
constant_integer (int: INTEGER): CONSTANT_INTEGER
        -- Reference to a integer constant with the value 'int'.
        -- Create integer constant if it does not exist.
    ensure
        integer_in_pool: has_constant_integer (int)
        result_not_void: Result /= void

constant_float (float: REAL): CONSTANT_FLOAT
        -- Reference to a float constant with the value 'float'.
        -- Create float constant if it does not exist.
    ensure
        float_in_pool: has_constant_float (float)
        result_not_void: Result /= void

constant_long (long: INTEGER_64): CONSTANT_LONG
        -- Reference to a long constant with the value 'long'.
        -- Create long constant if it does not exist.
    ensure
        long_in_pool: has_constant_long (long)
        result_not_void: Result /= void

constant_double (double: DOUBLE): CONSTANT_DOUBLE
        -- Reference to a double constant with the value 'double'.
        -- Create double constant if it does not exist.
    ensure
        double_in_pool: has_constant_double (double)
        result_not_void: Result /= void

constant_name_and_type
    (name, type: STRING): CONSTANT_NAME_AND_TYPE
        -- Reference to a name and type constant with the 'name' and the 'type'.
        -- Create name and type and UTF-8 constants if they do not exist.
    require
        name_not_void: name /= void
        type_not_void: type /= void
    ensure
        name_and_type_in_pool:
            has_constant_name_and_type (name, type)
        result_not_void: Result /= void

constant_utf8 (string: STRING): CONSTANT_UTF8
        -- Reference to a UTF-8 constant with the given 'string'.
        -- Create such a constant if it does not exist.
    require
        string_not_void: string /= void
        string_not_empty: not string.is_empty
```

```
        ensure
            utf8_in_pool: has_constant_utf8 (string)
            result_not_void: Result /= void

    constant_utf8_unicode
        (string: ARRAYED_LIST [INTEGER_16]): CONSTANT_UTF8
            -- Reference to a UTF-8 constant with the given 'string'.
            -- Create such a constant if it does not exist.
        require
            string_not_void: string /= void
            string_not_empty: not string.is_empty
        ensure
            utf8_in_pool: has_constant_utf8_unicode (string)
            result_not_void: Result /= void

    constant_pool: CONSTANT_POOL [CONSTANT]
            -- Generated constant pool

    const: JAVA_CONSTANTS
            -- Reference to the java constants object

feature -- Status report

    has_constant_class (name: STRING): BOOLEAN
            -- Is a class constant with the 'name' in the constant pool?
        require
            name_not_void: name /= void
            name_not_emtpy: not name.is_empty

    has_constant_field_ref
        (class_name, name, descriptor: STRING): BOOLEAN
            -- Is a field ref constant with the class 'class_name',
            -- the field name 'name' and the 'descriptor' in the constant pool?
        require
            class_name_not_void: class_name /= void
            name_not_void: name /= void

    has_constant_method_ref
        (class_name, name, descriptor: STRING): BOOLEAN
            -- Is a method ref constant with the class 'class_name',
            -- the method name 'name' and the 'descriptor' in the constant pool?
        require
            class_name_not_void: class_name /= void
            name_not_void: name /= void

    has_constant_interface_method_ref
        (class_name, name, descriptor: STRING): BOOLEAN
```

```
        -- Is a interface method ref constant with the class 'class_name',
        -- the interface method name 'name' and the 'descriptor' in the
        -- constant pool?
    require
        class_name_not_void: class_name /= void
        name_not_void: name /= void

has_constant_string (string: STRING): BOOLEAN
        -- Is a string constant with content 'string' in the constant pool?
    require
        string_not_void: string /= void
        string_not_empty: not string.is_empty

has_constant_string_unicode
    (string: ARRAYED_LIST [INTEGER_16]): BOOLEAN
        -- Is a string constant with content 'string' in the constant pool?
    require
        string_not_void: string /= void
        string_not_empty: not string.is_empty

has_constant_integer (int: INTEGER): BOOLEAN
        -- Is an integer constant with the value 'int' in the constant pool?

has_constant_float (float: REAL): BOOLEAN
        -- Is a float constant with the value 'float' in the constant pool?

has_constant_long (long: INTEGER_64): BOOLEAN
        -- Is a long constant with the value 'long' in the constant pool?

has_constant_double (double: DOUBLE): BOOLEAN
        -- Is a double constant with the value 'double' in the constant pool?

has_constant_name_and_type (name, type: STRING): BOOLEAN
        -- Is a name and type constant with the 'name' and the 'type'
        -- in the constant pool?
    require
        name_not_void: name /= void
        type_not_void: type /= void

has_constant_utf8 (string: STRING): BOOLEAN
        -- Is a UTF-8 constant with content 'string' in the constant pool?
    require
        string_not_void: string /= void
        string_not_empty: not string.is_empty

has_constant_utf8_unicode
    (string: ARRAYED_LIST [INTEGER_16]): BOOLEAN
```

```
        -- Is a UTF-8 constant with content 'string' in the constant pool?
      require
        string_not_void: string /= void
        string_not_empty: not string.is_empty

  invariant

    constant_pool_not_void: constant_pool /= void
    const_not_void: const /= void

  end
```

Class: FIELD_GENERATOR

Creates a field and sets its content.

```
  class
    FIELD_GENERATOR

  create
    make_from_constant_pool,
    make_from_constant_pool_gen

  feature -- Initialization

    make_from_constant_pool (pool: CONSTANT_POOL [CONSTANT])
        -- Create field generator object that uses the constant pool 'pool'.
      require
        pool_not_void: pool /= void
      ensure
        constant_pool_assigned: constant_pool_gen.constant_pool = pool

    make_from_constant_pool_gen (gen: CONSTANT_POOL_GENERATOR)
        -- Create field generator object that uses the constant pool
        -- generator 'gen'.
      require
        gen_not_void: gen /= void
      ensure
        gen_assigned: constant_pool_gen = gen

  feature -- Access

    field: FIELD
        -- Generated field

    constant_pool_gen: CONSTANT_POOL_GENERATOR
        -- Generator for the constant pool used in the field
```

```
feature -- Element change

    add_attribute (attr: FIELD_ATTRIBUTE)
            -- Add the attribute 'attr' to the field.
        require
            attr_not_void: attr /= void
        ensure
            attributes_not_void: field.attributes /= void
            attr_added: field.attributes.has (attr)

    set_access_flags (flags: INTEGER)
            -- Set the access flags of the field to 'flags'.
        require
            valid_flags: field.access_flags.valid_flags (flags)
        ensure
            access_flags_assigned: field.access_flags.access_flags = flags

    set_name (name: STRING)
            -- Set name of the field to 'name'.
        require
            name_not_void: name /= void
            name_not_empty: not name.is_empty
        ensure
            name_index_not_void: field.name_index /= void
            name_assigned: field.name_index.value.is_equal (name)

    set_descriptor (desc: STRING)
            -- Set the decriptor of the field to 'desc'.
        require
            desc_not_void: desc /= void
            desc_not_empty: not desc.is_empty
        ensure
            descriptor_index_not_void: field.descriptor_index /= void
            desc_assigned: field.descriptor_index.value.is_equal (desc)

invariant

    field_not_void: field /= void
    field_flags_not_void: field.access_flags /= void
    constant_pool_gen_not_void: constant_pool_gen /= void

end
```

Class: METHOD_GENERATOR

Creates a method and sets its content.

```
class
    METHOD_GENERATOR

create
    make_from_constant_pool,
    make_from_constant_pool_gen

feature -- Initialization

    make_from_constant_pool (pool:  CONSTANT_POOL [CONSTANT])
            -- Create method generator object that uses the constant pool 'pool'.
        require
            pool_not_void: pool /= void
        ensure
            pool_assigned: constant_pool_gen.constant_pool = pool

    make_from_constant_pool_gen (gen:  CONSTANT_POOL_GENERATOR)
            -- Create method generator object that uses the constant pool
            -- generator 'gen'.
        require
            gen_not_void: gen /= void
        ensure
            gen_assigend: constant_pool_gen = gen

feature -- Access

    method: METHOD
            -- Generated method

    constant_pool_gen: CONSTANT_POOL_GENERATOR
            -- Generator for the constant pool used in the method

    exception_attribute: EXCEPTIONS_ATTRIBUTE
            -- Exceptions that the method may throw

    const: JAVA_CONSTANTS
            -- Reference to the java constants object

feature -- Element change

    add_attribute (attr:  METHOD_ATTRIBUTE)
            -- Add the attribute 'attr' to the method.
        require
            attr_not_void: attr /= void
```

```
    ensure
      attributes_not_void: method.attributes /= void
      attr_added: method.attributes.has (attr)

  add_code (code: CODE_ATTRIBUTE)
      -- Add the code attribute 'code' to the method. It updates
      -- the local variable count of 'code', if the parameters need
      -- more local varaibles than the code uses.
    require
      code_not_void: code /= void
      descriptor_not_void: method.descriptor_index /= void
    ensure
      attributes_not_void: method.attributes /= void
      code_added: method.attributes.has (code)

  set_access_flags (flags: INTEGER)
      -- Set the access flags of the method to 'flags'.
    require
      valid_flags: method.access_flags.valid_flags (flags)
    ensure
      access_flags_assigned: method.access_flags.access_flags = flags

  set_name (name: STRING)
      -- Set the name of the method to 'name'.
    require
      name_not_void: name /= void
      name_not_empty: not name.is_empty
    ensure
      name_index_not_void: method.name_index /= void
      name_assigned: method.name_index.value.is_equal (name)

  set_descriptor (desc: STRING)
      -- Set the descriptor of the method to 'desc'.
    require
      desc_not_void: desc /= void
      desc_not_empty: not desc.is_empty
    ensure
      descriptor_index_not_void: method.descriptor_index /= void
      desc_assigned: method.descriptor_index.value.is_equal (desc)

  add_exception (name: STRING)
      -- Add the exception 'name' that the method may throw.
    require
      name_not_void: name /= void
      name_not_empty: not name.is_empty
    ensure
      attributes_not_void: method.attributes /= void
```

```
invariant

    method_not_void: method /= void
    method_flags_not_void: method.access_flags /= void
    constant_pool_gen_not_void: constant_pool_gen /= void
    const_not_void: const /= void

end
```

A.2.2 Cluster: code_blocks

Class: BLOCK

Block of JVM code.

```
deferred class
    BLOCK

feature -- Access

    first_instruction: INSTRUCTION
            -- First instruction in the block
        require
            is_complete: is_complete

    last_instruction: INSTRUCTION
            -- Last instruction in the block
        require
            is_complete: is_complete

    instructions: LINKED_LIST [INSTRUCTION]
            -- Code of the current block without any address computing
        require
            is_complete: is_complete
        ensure
            result_not_void: Result /= void

    code_attribute (cpg:  CONSTANT_POOL_GENERATOR): CODE_ATTRIBUTE
            -- Create (using 'cpg') and return the code attribute of the current block.
            -- If the block changes, an earlier created attribute will not be updated.
        require
            cpg_not_void: cpg /= void
            is_complete: is_complete

    max_stack: INTEGER
            -- Maximal stack depth. If it is user-defined, return the user value,
```

```
            -- else look in the code attribute.
        require
            is_complete: is_complete
        ensure
            user_set_max_stack: is_max_stack_user_set implies
                Result = user_max_stack
            computed_max_stack: not is_max_stack_user_set implies
                Result = max_stack_computed

    end_stack: INTEGER
            -- Stack depth at the end of the block. If it is user-defined, return
            -- the user value, else compute it.
        require
            is_complete: is_complete
        ensure
            user_set_end_stack: is_end_stack_user_set implies
                Result = user_end_stack
            computed_end_stack: not is_end_stack_user_set implies
                Result = end_stack_computed

    max_locals: INTEGER
            -- Maximal number of used local variables
        require
            is_complete: is_complete

    exception_table: CLASS_FILE_TABLE [EXCEPTION_INFO]
            -- Content of the exception table of this block
        require
            is_complete: is_complete
        ensure
            no_empty_result: Result /= void implies not Result.is_empty

    line_number_table: CLASS_FILE_TABLE [LINE_NUMBER_INFO]
            -- Content of the line number table attribute of this block
        require
            is_complete: is_complete
        ensure
            no_empty_result: Result /= void implies not Result.is_empty

    local_variable_table: CLASS_FILE_TABLE [LOCAL_VARIABLE_INFO]
            -- Content of the local variable table attribute of this block
        require
            is_complete: is_complete
        ensure
            no_empty_result: Result /= void implies not Result.is_empty

feature -- Status report
```

```
    is_return_block: BOOLEAN
            -- Does the block return in every case to the caller?
        require
            is_complete: is_complete

    is_complete: BOOLEAN
            -- Has the block enough information to work with?

    is_max_stack_user_set: BOOLEAN
            -- Is the the maximal stack depth user defined?

    is_end_stack_user_set: BOOLEAN
            -- Is the the stack depth at the end of the block user defined?

feature -- Status setting

    compute_max_stack is
            -- Set is_max_stack_user_set to false.
        ensure
            not_is_max_stack_user_set: not is_max_stack_user_set

    compute_end_stack is
            -- Set is_end_stack_user_set to false.
        ensure
            not_is_end_stack_user_set: not is_end_stack_user_set

feature -- Element change

    set_max_stack (depth:  INTEGER)
            -- Set 'user_max_stack' to 'depth' and mark it as user defined.
        ensure
            is_max_stack_user_set: is_max_stack_user_set
            max_stack_assigned: max_stack = depth

    set_end_stack (depth:  INTEGER)
            -- Set 'user_end_stack' to 'depth' and mark it as user defined.
        ensure
            is_end_stack_user_set: is_end_stack_user_set
            end_stack_assigned: end_stack = depth

    add_line_number_info (lni:  LINE_NUMBER_INFO)
            -- Add the line number info 'lni' to the block.
        require
            lni_not_void: lni /= void
        ensure
            line_number_info_added: line_number_table.has (lni)
```

```
    add_local_variable_info (lvi:  LOCAL_VARIABLE_INFO)
            -- Add the local variable info 'lvi' to the block.
        require
            lvi_not_void: lvi /= void
        ensure
            local_variable_info_added: local_variable_table.has (lvi)

end
```

Class: CODE_BLOCK

Linear section of JVM code. The code block must not contain jump instructions.

```
class
    CODE_BLOCK

inherit
    BLOCK
    LINKED_LIST[INSTRUCTION]

create
    make

feature -- Initialization

    make is
            -- Create a new code block object.

feature -- Access

    first_instruction: INSTRUCTION
            -- First instruction in the block
        ensure then
            definition: Result = first

    last_instruction: INSTRUCTION
            -- Last instruction in the block
        ensure then
            definition: Result = last

    instructions: LINKED_LIST [INSTRUCTION]
            -- Code of the current block without any address computing
        ensure then
            definition: Result = Current

    max_locals: INTEGER
```

```
        -- Maximal number of used local variables

    exception_table: CLASS_FILE_TABLE [EXCEPTION_INFO]
        -- Content of the exception table of this block

    line_number_table: CLASS_FILE_TABLE [LINE_NUMBER_INFO]
        -- Content of the line number table attribute of this block

    local_variable_table: CLASS_FILE_TABLE [LOCAL_VARIABLE_INFO]
        -- Content of the local variable table attribute of this block

    const: JAVA_CONSTANTS
        -- Reference to the java constants object

feature -- Status report

    is_return_block: BOOLEAN
        -- Does the block return in every case to the caller?

    is_complete: BOOLEAN
        -- Has the block enough information to work with?
      ensure then
        definition: Result = not is_empty and then
          not (is_max_stack_user_set and is_end_stack_user_set)
          implies not is_jump_instruction

feature -- Status setting

    return_block is
        -- Block returns in every case to the caller.
      ensure
        definition: is_return_block

feature -- Element change

    add_line_number_info (lni:  LINE_NUMBER_INFO)
        -- Add the line number info 'lni' to the block.

    add_local_variable_info (lvi:  LOCAL_VARIABLE_INFO)
        -- Add the local variable info 'lvi' to the block.

invariant

    exceptions_table_is_void: exception_table = void

end
```

Class: CONCAT_BLOCK

Concatenates two blocks.

class
CONCAT_BLOCK

inherit
BLOCK

feature -- Access

first_block: BLOCK
-- First block

second_block: BLOCK
-- Second block that is concatenated to the first

first_instruction: INSTRUCTION
-- First instruction in the block
ensure then
definition: Result = first_block.first_instruction

last_instruction: INSTRUCTION
-- Last instruction in the block
ensure then
definition: Result = second_block.last_instruction

instructions: LINKED_LIST [INSTRUCTION]
-- Code of the current block without any address computing

max_locals: INTEGER
-- Maximal number of used local variables
ensure then
greater_or_equal_first_block: Result >= first_block.max_locals
greater_or_equal_second_block:
Result >= second_block.max_locals

exception_table: CLASS_FILE_TABLE [EXCEPTION_INFO]
-- Content of the exception table of this block

line_number_table: CLASS_FILE_TABLE [LINE_NUMBER_INFO]
-- Content of the line number table attribute of this block

local_variable_table: CLASS_FILE_TABLE [LOCAL_VARIABLE_INFO]
-- Content of the local variable table attribute of this block

feature -- Status report

```
    is_return_block: BOOLEAN
        -- Does the block return in every case to the caller?
      ensure then
        definition: Result = second_block.is_return_block

    is_complete: BOOLEAN
        -- Has the block enough information to work with?
      ensure then
        definition: Result = (first_block /= void and then
          first_block.is_complete and then
          second_block /= void and then second_block.is_complete)

  feature -- Element change

    set_first_block (block:  like first_block)
        -- Set 'first_block' to 'block'.
      require
        block_not_void: block /= void
      ensure
        block_assigned: first_block = block

    set_second_block (block:  like second_block)
        -- Set 'second_block' to 'block'.
      require
        block_not_void: block /= void
      ensure
        block_assigned: second_block = block

    add_line_number_info (lni:  LINE_NUMBER_INFO)
        -- Add the line number info 'lni' to the block.

    add_local_variable_info (lvi:  LOCAL_VARIABLE_INFO)
        -- Add the local variable info 'lvi' to the block.

  end
```

Class: DO_WHILE_BLOCK

Block structure of a do-while loop.

```
  class
    DO_WHILE_BLOCK

  inherit
    BLOCK
```

```
create
  make

feature -- Initialization

  make is
      -- Create a new do while block object.

feature -- Access

  test_branch: INSTRUCTION
      -- Test and branch operation

  loop_body: BLOCK
      -- Body of the loop

  const: JAVA_CONSTANTS
      -- Reference to the java constants object

  first_instruction: INSTRUCTION
      -- First instruction in the block
    ensure then
      definition: Result = loop_body.first_instruction

  last_instruction: INSTRUCTION
      -- Last instruction in the block
    ensure then
      definition: Result = test_branch

  instructions: LINKED_LIST [INSTRUCTION]
      -- Code of the current block without any address computing

  max_locals: INTEGER
      -- Maximal number of used local variables
    ensure then
      definition: Result = loop_body.max_locals

  exception_table: CLASS_FILE_TABLE [EXCEPTION_INFO]
      -- Content of the exception table of this block

  line_number_table: CLASS_FILE_TABLE [LINE_NUMBER_INFO]
      -- Content of the line number table attribute of this block

  local_variable_table: CLASS_FILE_TABLE [LOCAL_VARIABLE_INFO]
      -- Content of the local variable table attribute of this block

feature -- Status report
```

```
    Is_return_block: BOOLEAN is False
            -- Does the block return in every case to the caller?

    is_complete: BOOLEAN
            -- Has the block enough information to work with?
        ensure then
            definition: Result = (test_branch /= void and then
                loop_body /= void and then loop_body.is_complete)

feature -- Element change

    set_test_branch (opcode:  INTEGER)
            -- Set the test and branch operation to 'opcode'.
        require
            valid_opcode: (opcode >= const.ifeq and
                opcode <= const.if_acmpne) or
                (opcode >= const.ifnull and opcode <= const.ifnonnull)
        ensure
            test_branch_not_void: test_branch /= void
            test_branch_assigned: test_branch.opcode = opcode

    set_loop_body (body:  like loop_body)
            -- Set 'loop_body' to 'body'.
        require
            body_not_void: body /= void
        ensure
            loop_body_assigned: loop_body = body

    add_line_number_info (lni:  LINE_NUMBER_INFO)
            -- Add the line number info 'lni' to the block.

    add_local_variable_info (lvi:  LOCAL_VARIABLE_INFO)
            -- Add the local variable info 'lvi' to the block.

invariant

    const_not_void: const /= void

end
```

Class: IF_THEN_BLOCK

Block structure for if-then.

```
class
    IF_THEN_BLOCK
```

```
inherit
    BLOCK

create
    make

feature -- Initalization

    make is
            -- Create a new if then block object.

feature -- Access

    test_branch: INSTRUCTION
            -- Test and branch operation, branch target is not set

    then_body: BLOCK
            -- Block that is executed if condition is true

    after_if: BLOCK
            -- Block that follows the if-then construct

    const: JAVA_CONSTANTS
            -- Reference to the java constants object

    first_instruction: INSTRUCTION
            -- First instruction in the block
        ensure then
            definition: Result = test_branch

    last_instruction: INSTRUCTION
            -- Last instruction in the block
        ensure then
            definition: Result = after_if.last_instruction

    instructions: LINKED_LIST [INSTRUCTION]
            -- Code of the current block without any address computing

    max_locals: INTEGER
            -- Maximal number of used local variables
        ensure then
            greater_or_equal_then_body: Result >= then_body.max_locals
            greater_or_equal_after_if: Result >= after_if.max_locals

    exception_table: CLASS_FILE_TABLE [EXCEPTION_INFO]
            -- Content of the exception table of this block
```

```
    line_number_table: CLASS_FILE_TABLE [LINE_NUMBER_INFO]
        -- Content of the line number table attribute of this block

    local_variable_table: CLASS_FILE_TABLE [LOCAL_VARIABLE_INFO]
        -- Content of the local variable table attribute of this block

feature -- Status report

    is_return_block: BOOLEAN
        -- Does the block return in every case to the caller?
      ensure then
        definition: Result = after_if.is_return_block

    is_complete: BOOLEAN
        -- Has the block enough information to work with?
      ensure then
        definition: Result = (test_branch /= void and then
          then_body /= void and then then_body.is_complete and then
          after_if /= void and then after_if.is_complete)

feature -- Element change

    set_test_branch (opcode:  INTEGER)
        -- Set test and branch operation to 'opcode'.
      require
        valid_opcode: (opcode >= const.ifeq and
          opcode <= const.if_acmpne) or
          (opcode >= const.ifnull and opcode <= const.ifnonnull)
      ensure
        test_branch_not_void: test_branch /= void
        test_branch_assigned: test_branch.opcode = opcode

    set_then_body (body:  like then_body)
        -- Set 'then_body' to 'body'.
      require
        body_not_void: body /= void
      ensure
        then_body_assigned: then_body = body

    set_after_if (after:  like after_if)
        -- Set 'after_if' to 'after'.
      require
        after_not_void: after /= void
      ensure
        after_if_assigned: after_if = after
```

add_line_number_info (lni: LINE_NUMBER_INFO)
-- Add the line number info 'lni' to the block.

add_local_variable_info (lvi: LOCAL_VARIABLE_INFO)
-- Add the local variable info 'lvi' to the block.

invariant

const_not_void: const /= void

end

Class: IF_THEN_ELSE_BLOCK

Block structure for if-then-else.

class
IF_THEN_ELSE_BLOCK

inherit
BLOCK

create
make

feature -- Initialization

make is
-- Create a new if then else block object.

feature -- Access

test_branch: INSTRUCTION
-- Test and branch operation, branch target not set

then_body: BLOCK
-- Block that is executed if condition is true.

else_body: BLOCK
-- Block that is executed if condition is false.

after_if: BLOCK
-- Block that follows the if-then-else consturct.

const: JAVA_CONSTANTS
-- Reference to the java constants object

```
        first_instruction: INSTRUCTION
                -- First instruction in the block
            ensure then
                definition: Result = test_branch

        last_instruction: INSTRUCTION
                -- Last instruction in the block

        instructions: LINKED_LIST [INSTRUCTION]
                -- Code of the current block without any address computing

        max_locals: INTEGER
                -- Maximal number of used local variables
            ensure then
                greater_or_equal_then_body: Result >= then_body.max_locals
                greater_or_equal_else_body: Result >= else_body.max_locals

        exception_table: CLASS_FILE_TABLE [EXCEPTION_INFO]
                -- Content of the exception table of this block

        line_number_table: CLASS_FILE_TABLE [LINE_NUMBER_INFO]
                -- Content of the line number table attribute of this block

        local_variable_table: CLASS_FILE_TABLE [LOCAL_VARIABLE_INFO]
                -- Content of the local variable table attribute of this block

    feature -- Status report

        is_return_block: BOOLEAN
                -- Does the block return in every case to the caller?
            ensure then
                definition: Result = (then_body.is_return_block and then
                    else_body.is_return_block) or else
                    after_if.is_return_block

        is_complete: BOOLEAN
                -- Has the block enough information to work with?
            ensure then
                definition: Result = (test_branch /= void and then
                    then_body /= void and then then_body.is_complete and then
                    else_body /= void and then else_body.is_complete and then
                    ((then_body.is_return_block and then
                    else_body.is_return_block) or else
                    (after_if /= void and then after_if.is_complete)))

    feature -- Element change
```

```
        set_test_branch (opcode: INTEGER)
                -- Set test and branch operation to 'opcode'.
            require
                valid_opcode: (opcode >= const.ifeq and
                    opcode <= const.if_acmpne) or
                    (opcode >= const.ifnull and opcode <= const.ifnonnull)
            ensure
                test_branch_not_void: test_branch /= void
                test_branch_assigned: test_branch.opcode = opcode

        set_then_body (body: like then_body)
                -- Set 'then_body' to 'body'.
            require
                body_not_void: body /= void
            ensure
                then_body_assigned: then_body = body

        set_else_body (body: like else_body)
                -- Set 'else_body' to 'body'.
            require
                body_not_void: body /= void
            ensure
                else_body_assigned: else_body = body

        set_after_if (after: like after_if)
                -- Set 'after_if' to 'after'.
            require
                after_not_void: after /= void
            ensure
                after_if_assigned: after_if = after

        add_line_number_info (lni: LINE_NUMBER_INFO)
                -- Add the line number info 'lni' to the block.

        add_local_variable_info (lvi: LOCAL_VARIABLE_INFO)
                -- Add the local variable info 'lvi' to the block.

    invariant

        const_not_void: const /= void

    end
```

Class: PRINT_BLOCK

Block that writes to the standard output.

```
class
    PRINT_BLOCK

inherit
    BLOCK

create
    make_from_constant_pool,
    make_from_constant_pool_gen

feature -- Initialization

    make_from_constant_pool (cp:  CONSTANT_POOL [CONSTANT])
            -- Create a new print block object using the constant pool 'cp'.
        require
            cp_not_void: cp /= void
        ensure
            is_whole_line: is_whole_line
            is_write_string: is_write_string

    make_from_constant_pool_gen (gen:  CONSTANT_POOL_GENERATOR)
            -- Create a new print block object using the constant pool generator 'gen'.
        require
            gen_not_void: gen /= void
        ensure
            is_whole_line: is_whole_line
            is_write_string: is_write_string

feature -- Access

    const: JAVA_CONSTANTS
            -- Reference to the java constants object

    cp_gen: CONSTANT_POOL_GENERATOR
            -- Constant pool generator used in the print generator

    first_instruction: INSTRUCTION
            -- First instruction in the block

    last_instruction: INSTRUCTION
            -- Last instruction in the block

    instructions: LINKED_LIST [INSTRUCTION]
            -- Code of the current block without any address computing

    Max_locals: INTEGER is 0
            -- Maximal number of used local variables
```

exception_table: `CLASS_FILE_TABLE [EXCEPTION_INFO]`
-- Content of the exception table of this block

line_number_table: `CLASS_FILE_TABLE [LINE_NUMBER_INFO]`
-- Content of the line number table attribute of this block

local_variable_table: `CLASS_FILE_TABLE [LOCAL_VARIABLE_INFO]`
-- Content of the local variable table attribute of this block

feature -- Status report

is_write_string: `BOOLEAN`
-- Does block write a string?

is_write_object: `BOOLEAN`
-- Does block write an object?

is_write_int: `BOOLEAN`
-- Does block write an int value?

is_write_float: `BOOLEAN`
-- Does block write a float value?

is_write_double: `BOOLEAN`
-- Does block write a double value?

is_write_long: `BOOLEAN`
-- Does block write a long value?

is_write_char: `BOOLEAN`
-- Does block write a char value?

is_write_boolean: `BOOLEAN`
-- Does block write a boolean value?

is_write_char_array: `BOOLEAN`
-- Does block write a char array?

is_write_line_break: `BOOLEAN`
-- Does block only write a line break?

is_whole_line: `BOOLEAN`
-- Does block write a line break after the written value (println)?

Is_return_block: `BOOLEAN is False`
-- Does the block return in every case to the caller?

```
    Is_complete: BOOLEAN is True
            -- Has the block enough information to work with?

feature -- Status setting

    write_string is
            -- Block writes a string.
        ensure
            definition: is_write_string = True

    write_object is
            -- Block writes an object.
        ensure
            definition: is_write_object = True

    write_int is
            -- Block writes an int value.
        ensure
            definition: is_write_int = True

    write_float is
            -- Block writes a float value.
        ensure
            definition: is_write_float = True

    write_double is
            -- Block writes a double value.
        ensure
            definition: is_write_double = True

    write_long is
            -- Block writes a long value.
        ensure
            definition: is_write_long = True

    write_char is
            -- Block writes a char value.
        ensure
            definition: is_write_char = True

    write_boolean is
            -- Block writes a boolean value.
        ensure
            definition: is_write_boolean = True

    write_char_array is
```

```
        -- Block writes a char array.
      ensure
        definition: is_write_char_array = True

    write_line_break is
        -- Block only writes a line break.
      ensure
        definition: is_write_line_break = True

    whole_line (b:  BOOLEAN)
        -- Block writes a line break after the written value (println / print).
      ensure
        definition: is_whole_line = b

  feature -- Element change

    add_line_number_info (lni:  LINE_NUMBER_INFO)
        -- Add the line number info 'lni' to the block.

    add_local_variable_info (lvi:  LOCAL_VARIABLE_INFO)
        -- Add the local variable info 'lvi' to the block.

  invariant

    const_not_void: const /= void
    status_set: status /= 0
    only_string: is_write_string implies status.bit_xor (w_string) = 0
    only_object: is_write_object implies status.bit_xor (w_object) = 0
    only_int: is_write_int implies status.bit_xor (w_int) = 0
    only_float: is_write_float implies status.bit_xor (w_float) = 0
    only_double: is_write_double implies status.bit_xor (w_double) = 0
    only_long: is_write_long implies status.bit_xor (w_long) = 0
    only_char: is_write_char implies status.bit_xor (w_char) = 0
    only_boolean: is_write_boolean implies
      status.bit_xor (w_boolean) = 0
    only_char_array: is_write_char_array implies
      status.bit_xor (w_char_array) = 0
    only_line_break: is_write_line_break implies
      status.bit_xor (w_line_break) = 0
    first_instruction_not_void: first_instruction /= void
    last_instruction_not_void: last_instruction /= void
    cp_gen_not_void: cp_gen /= void

  end
```

Class: SWITCH_BLOCK

Block structure of a switch statement.

class
SWITCH_BLOCK

inherit
BLOCK

create
make

feature -- Initialization

make is
-- Create a new switch block object.
ensure
no_values_inserted: min_value > max_value

feature -- Access

switch_instruction: INSTRUCTION
-- Instruction that implements the switch.

first_instruction: INSTRUCTION
-- First instruction in the block
ensure then
definition: Result = switch_instruction

last_instruction: INSTRUCTION
-- Last instruction in the block

instructions: LINKED_LIST [INSTRUCTION]
-- Code of the current block without any address computing

const: JAVA_CONSTANTS
-- Reference to the java constants object

case_blocks: ARRAYED_LIST [TUPLE [INTEGER, BOOLEAN, BLOCK, BOOLEAN]]
-- Labels with the value or default, and a code block that may
-- contain a break.

after_switch: BLOCK
-- Block that follows the switch statement.

max_locals: INTEGER
-- Maximal number of used local variables

```
    exception_table: CLASS_FILE_TABLE [EXCEPTION_INFO]
        -- Content of the exception table of this block

    line_number_table: CLASS_FILE_TABLE [LINE_NUMBER_INFO]
        -- Content of the line number table attribute of this block

    local_variable_table: CLASS_FILE_TABLE [LOCAL_VARIABLE_INFO]
        -- Content of the local variable table attribute of this block

feature -- Status report

    is_return_block: BOOLEAN
        -- Does the block return in every case to the caller?
      ensure then
        definition: Result = ((after_switch /= void and then
          after_switch.is_return_block) or else
          (after_switch = void and then are_all_case_return))

    is_complete: BOOLEAN
        -- Has the block enough information to work with?
      ensure then
        definition: Result = (switch_instruction /= void and then
          are_case_blocks_complete and then
          (after_switch /= void implies
          after_switch.is_complete) and then
          (after_switch = void implies
          (is_default_inserted and then
          not has_label_at_end and then
          (are_all_case_return or else not is_one_case_break))))

    is_value_unique (value: INTEGER): BOOLEAN
        -- Is there no other case block with the 'value'?

    is_default_inserted: BOOLEAN
        -- Is there already a default label inserted?

    has_label_at_end: BOOLEAN
        -- Does switch contain labels at the end that are not followed by a block?

feature -- Element change

    set_switch_instruction (opcode: INTEGER)
        -- Set switch instruction to 'opcode' (tableswitch, lookupswitch).
      require
        valid_opcode: opcode = const.tableswitch or else
          opcode = const.lookupswitch
```

```
        ensure
            switch_instruction_not_void: switch_instruction /= void
            switch_instruction_assigned: switch_instruction.opcode = opcode

        set_after_switch (block: BLOCK)
            -- Set 'after_swithc' to 'block'.
        require
            block_not_void: block /= void
        ensure
            after_switch_assigend: after_switch = block

        add_case_block (value: INTEGER; is_default: BOOLEAN;
        block: BLOCK; has_break: BOOLEAN)
            -- Add a label with the 'value' or default 'is_defaut' that may have
            -- a code block 'block' that may contain a break statement 'has_break'.
        require
            at_most_one_default: is_default implies
                not is_default_inserted
            is_value_unique: not is_default implies is_value_unique (value)
        ensure
            case_blocks_not_empty: not case_blocks.is_empty
            default_inserted: is_default implies is_default_inserted

        add_line_number_info (lni: LINE_NUMBER_INFO)
            -- Add the line number info 'lni' to the block.

        add_local_variable_info (lvi: LOCAL_VARIABLE_INFO)
            -- Add the local variable info 'lvi' to the block.

    invariant

        const_not_void: const /= void
        case_blocks_not_void: case_blocks /= void

    end
```

Class: TRY_CATCH_BLOCK

Block structure of try-catch.

```
    class
        TRY_CATCH_BLOCK

    inherit
        BLOCK

    create
```

```
    make

feature -- Initialization

    make is
            -- Create a new try catch block object.

feature -- Access

    try_body: BLOCK
            -- Try body in the try-catch construct

    exception_handlers: ARRAYED_LIST [TUPLE [CONSTANT_CLASS, BLOCK]]
            -- Catched exceptions with the corresponding handler

    after_try_catch: BLOCK
            -- Block that follows the try-catch construct

    const: JAVA_CONSTANTS
            -- Rerefernce to the java constants object

    first_instruction: INSTRUCTION
            -- First instruction in the block
        ensure then
            definition: Result = try_body.first_instruction

    last_instruction: INSTRUCTION
            -- Last instruction in the block

    instructions: LINKED_LIST [INSTRUCTION]
            -- Code of the current block without any address computing

    max_locals: INTEGER
            -- Maximal number of used local variables

    exception_table: CLASS_FILE_TABLE [EXCEPTION_INFO]
            -- Content of the exception table of this block

    line_number_table: CLASS_FILE_TABLE [LINE_NUMBER_INFO]
            -- Content of the line number table attribute of this block

    local_variable_table: CLASS_FILE_TABLE [LOCAL_VARIABLE_INFO]
            -- Content of the local variable table attribute of this block

feature -- Status report

    is_return_block: BOOLEAN
```

```
        -- Does the block return in every case to the caller?
    ensure then
        definition: Result = are_all_return_blocks or else
            after_try_catch.is_return_block

  is_complete: BOOLEAN
        -- Has the block enough information to work with?
    ensure then
        definition: Result = (try_body /= void and then
            try_body.is_complete and then
            are_exception_handlers_complete and then
            (are_all_return_blocks or else
            (after_try_catch /= void and then
            after_try_catch.is_complete)))

feature -- Element change

  set_try_body (body: like try_body)
        -- Set 'try_body' to 'body'.
    require
        body_not_void: body /= void
    ensure
        try_body_assigned: try_body = body

  add_exception_handler (excep_type: CONSTANT_CLASS; handler: BLOCK)
        -- Add an exception handler 'hanlder' that catches the exception
        -- type 'excep_type'.
    require
        excep_type_not_void: excep_type /= void
        handler_not_void: handler /= void
    ensure
        excep_type_added:
            exception_handlers.last.item (1) = excep_type
        handler_addes: exception_handlers.last.item (2) = handler

  set_after_try_catch (after: like after_try_catch)
        -- Set 'after_try_catch' to 'after'.
    require
        after_not_void: after /= void
    ensure
        after_try_catch_assigned: after_try_catch = after

  add_line_number_info (lni: LINE_NUMBER_INFO)
        -- Add the line number info 'lni' to the block.

  add_local_variable_info (lvi: LOCAL_VARIABLE_INFO)
        -- Add the local variable info 'lvi' to the block.
```

```
invariant

    exception_handlers_not_void: exception_handlers /= void
    const_not_void: const /= void

end
```

Class: WHILE_BLOCK

Block structure of a while loop.

```
class
    WHILE_BLOCK

inherit
    BLOCK

create
    make

feature -- Initialization

    make is
            -- Create a new while block object.

feature -- Access

    test_branch: INSTRUCTION
            -- Test and branch operation

    loop_body: BLOCK
            -- Body of the loop

    after_loop: BLOCK
            -- Block that is executed when the loop terminates

    const: JAVA_CONSTANTS
            -- Reference to the java constants object

    first_instruction: INSTRUCTION
            -- First instruction in the block
        ensure then
            definition: Result = test_branch

    last_instruction: INSTRUCTION
            -- Last instruction in the block
```

```
        ensure then
            definition: Result = after_loop.last_instruction

    instructions: LINKED_LIST [INSTRUCTION]
            -- Code of the current block without any address computing

    max_locals: INTEGER
            -- Maximal number of used local variables
        ensure then
            greater_or_equal_loop_body: Result >= loop_body.max_locals
            greater_or_equal_after_loop: Result >= after_loop.max_locals

    exception_table: CLASS_FILE_TABLE [EXCEPTION_INFO]
            -- Content of the exception table of this block

    line_number_table: CLASS_FILE_TABLE [LINE_NUMBER_INFO]
            -- Content of the line number table attribute of this block

    local_variable_table: CLASS_FILE_TABLE [LOCAL_VARIABLE_INFO]
            -- Content of the local variable table attribute of this block

feature -- Status report

    is_return_block: BOOLEAN
            -- Does the block return in every case to the caller?
        ensure then
            definition: Result = after_loop.is_return_block

    is_complete: BOOLEAN
            -- Has the block enough information to work with?
        ensure then
            definition: Result = (test_branch /= void and then
                loop_body /= void and then loop_body.is_complete and then
                after_loop /= void and then after_loop.is_complete)

feature -- Element change

    set_test_branch (opcode:  INTEGER)
            -- Set the test and branch operation to 'opcode'.
        require
            valid_opcode: (opcode >= const.ifeq and
                opcode <= const.if_acmpne) or (opcode >= const.ifnull and
                opcode <= const.ifnonnull)
        ensure
            test_branch_not_void: test_branch /= void
            test_branch_assigned: test_branch.opcode = opcode
```

```
        set_loop_body (body:  like loop_body)
                -- Set 'loop_body' to 'body'.
            require
                body_not_void: body /= void
            ensure
                loop_body_assigned: loop_body = body

        set_after_loop (after:  like after_loop)
                -- Set 'after_loop' to 'after'.
            require
                after_not_void: after /= void
            ensure
                after_loop_assigned: after_loop = after

        add_line_number_info (lni:  LINE_NUMBER_INFO)
                -- Add the line number info 'lni' to the block.

        add_local_variable_info (lvi:  LOCAL_VARIABLE_INFO)
                -- Add the local variable info 'lvi' to the block.

    invariant

        const_not_void: const /= void

    end
```

Appendix B

Java grammar

B.1 The Syntactic Grammar

Goal:
    CompilationUnit

B.2 Lexical Structure

Literal:
    IntegerLiteral
    FloatingPointLiteral
    BooleanLiteral
    CharacterLiteral
    StringLiteral
    NullLiteral

B.3 Types, Values, and Variables

Type:
    PrimitiveType
    ReferenceType

PrimitiveType:
    NumericType
    `boolean`

NumericType:
    IntegralType
    FloatingPointType

IntegralType: one of
`byte short int long char`

FloatingPointType: one of
`float double`

ReferenceType:
ClassOrInterfaceType
ArrayType

ClassOrInterfaceType:
Name

ClassType:
ClassOrInterfaceType

InterfaceType:
ClassOrInterfaceType

ArrayType:
PrimitiveType `[ ]`
Name `[ ]`
ArrayType `[ ]`

B.4 Names

Name:
SimpleName
QualifiedName

SimpleName:
Indentifier

QualifiedName:
Name . Identifier

B.5 Packages

CompilationUnit:
$PackageDeclaration_{opt}$ $ImportDeclarations_{opt}$ $TypeDeclarations_{opt}$

ImportDeclarations:
ImportDeclaration
ImportDeclarations ImportDeclaration

TypeDeclarations:
TypeDeclaration
TypeDeclarations TypeDeclaration

PackageDeclaration:
`package` *Name* `;`

ImportDeclaration:
SingleTypeImportDeclaration
TypeImportOnDemandDeclaration

SingleTypeImportDeclaration:
`import` *Name* `;`

TypeImportOnDemandDeclaration:
`import` *Name* `. * ;`

TypeDeclaration:
ClassDeclaration
InterfaceDeclaration
`;`

B.6 Productions Used Only in LALR(1) Grammar

Modifiers:
Modifier
Modifiers Modifier

Modifier: one of
`public protected private`
`static`
`abstract final native synchronized transient volatile`

B.7 Classes

B.7.1 Class Declaration

ClassDeclaration:

Modifiers$_{opt}$ `class` *Identifier Super*$_{opt}$ *Interfaces*$_{opt}$ *ClassBody*

Super:
`extends` *ClassType*

Interfaces:
`implements` *InterfaceTypeList*

InterfaceTypeList:
InterfaceType
InterfaceTypeList , *InterfaceType*

ClassBody:
{ *ClassBodyDeclarations*$_{opt}$ }

ClassBodyDeclarations:
ClassBodyDeclaration
ClassBodyDeclarations ClassBodyDeclaration

ClassBodyDeclaration:
ClassMemberDeclaration
StaticInitializer
ConstructorDeclaration

ClassMemberDeclaration:
FieldDeclaration
MethodDeclaration

B.7.2 Field Declarations

FieldDeclaration:
Modifiers$_{opt}$ *Type VariableDeclarators* ;

VariableDeclarators:
VariableDeclarator
VariableDeclarators , *VariableDeclarator*

VariableDeclarator:
VariableDeclaratorId
VariableDeclaratorId = *VariableInitializer*

VariableDeclaratorId:
Identifier
VariableDeclaratorId `[ ]`

VariableInitializer:
Expression
ArrayInitializer

B.7.3 Method Declarations

MethodDeclaration:
MethodHeader MethodBody

MethodHeader:
Modifiers$_{opt}$ *Type MethodDeclarator Throws*$_{opt}$
Modifiers$_{opt}$ `void` *MethodDeclarator Throws*$_{opt}$

MethodDeclarator:
Identifier `(` *FormalParamterList*$_{opt}$ `)`
MethodDeclarator `[ ]`

FormalParameterList:
FormalParameter
FormalParameterList , *FormalParameter*

FormalParameter:
Type VariableDeclaratorId

Throws:
`throws` *ClassTypeList*

ClassTypeList:
ClassType
ClassTypeList , *ClassType*

MethodBody:
Block
;

B.7.4 Static Initializers

StaticInitializer:
`static` *Block*

B.7.5 Constructor Declarations

ConstructorDeclaration:
Modifiers$_{opt}$ *ConstructorDeclarator Throws*$_{opt}$ *ConstructorBody*

ConstructorDeclarator:
SimpleName `(` *FormalParamterList*$_{opt}$ `)`

ConstructorBody:
`{` *ExplicitConstructorInvocation*$_{opt}$ *BlockStatements*$_{opt}$ `}`

ExplicitConstructorInvocation:
`this (` *ArgumentList*$_{opt}$ `) ;`
`super (` *ArgumentList*$_{opt}$ `) ;`

B.8 Interfaces

B.8.1 Interface Declarations

InterfaceDeclaration:
Modifiers$_{opt}$ `interface` *Identifier ExtendsInterfaces*$_{opt}$ *InterfaceBody*

ExtendsInterfaces:
`extends` *InterfaceType*
ExtendsInterfaces , *InterfaceType*

InterfaceBody;
`{` *InterfaceMemberDeclarations*$_{opt}$ `}`

InterfaceMemberDeclarations:
InterfaceMemberDeclaration
InterfaceMemberDeclarations InterfaceMemberDeclaration

InterfaceMemberDeclaration:
ConstantDeclaration
AbstractMethodDeclaration

ConstantDeclaration:
FieldDeclaration

AbstractMethodDeclaration:
MethodHeader ;

B.9 Arrays

ArrayInitializer:
{ *VariableInitializers*$_{opt}$,$_{opt}$ }

VariableInitializer:
VariableInitializer
VariableInitializers , *VariableInitializer*

B.10 Blocks and Statements

Block:
{ *BlockStatements*$_{opt}$ }

BlockStatements:
BlockStatement
BlockStatements BlockStatement

BlockStatement:
LocalVariableDeclarationStatement
Statement

LocalVariableDeclarationStatement:
LocalVariableDeclaration ;

LocalVariableDeclaration:
Type VariableDeclarators

Statement:
StatementWithoutTrailingSubstatement
LabeledStatement
IfThenStatement
IfThenElseStatement
WhileStatement
ForStatement

StatementNoShortIf:
StatementWithoutTrailingSubstatement
LabeledStatementNoShortIf
IfThenElseStatementNoShortIf
WhileStatementNoShortIf
ForStatementNoShortIf

StatementWithoutTrailingSubstatement:
Block
EmptyStatement
ExpressionStatement
SwitchStatement
DoStatement
BreakStatement
ContinueStatement
ReturnStatement
SynchronizedStatement
ThrowStatement
TryStatement

EmptyStatement:
`;`

LabeledStatement:
Indentifier `:` *Statement*

LabeledStatementNoShortIf:
Indentifier `:` *StatementNoShortIf*

ExpressionStatement:
StatementExpression `;`

StatementExpression:
Assignment
PreIncrementExpression
PreDecrementExpression
PostIncrementExpression
PostDecrementExpression
MethodInvocation
ClassInstanceCreationExpression

IfThenStatement:
`if (` *Expression* `)` *Statement*

IfThenElseStatement:
`if (` *Expression* `)` *StatementNoShortIf* `else` *Statement*

IfThenElseStatementNoShortIf:
`if (` *Expression* `)` *StatementNoShortIf* `else` *StatementNoShortIf*

SwitchStatement:
`switch (` *Expression* `)` *SwitchBlock*

SwitchBlock:

{ *SwitchBlockStatementGroups*$_{opt}$ *SwitchLabels*$_{opt}$ }

SwitchBlockStatementGroups:
SwitchBlockStatementGroup
SwitchBlockStatementGroups SwitchBlockStatementGroup

SwithcBlockStatementGroup:
SwitchLabels BlockStatements

SwitchLabels:
SwitchLabel
SwitchLabels SwitchLabel

SwitchLabel:
`case` *ConstantExpression* `:`
`default :`

WhileStatement:
`while (` *Expression* `)` *Statement*

WhileStatementNoShortIf:
`while (` *Expression* `)` *StatementNoShortIf*

DoStatement:
`do` *Statement* `while (` *Expression* `) ;`

ForStatement:
`for (` *ForInit*$_{opt}$ `;` *Expression*$_{opt}$ `;` *ForUpdate*$_{opt}$ `)` *Statement*

ForStatementNoShortIf:
`for (` *ForInit*$_{opt}$ `;` *Expression*$_{opt}$ `;` *ForUpdate*$_{opt}$ `)` *StatementNoShortIf*

ForInit:
StatementExpressionList
LocalVariableDeclaration

ForUpdate:
StatementExpressionList

StatementExpressionList:
StatementExpression
StatementExpressionList , StatementExpression

BreakStatement:
`break` *Identifier*$_{opt}$ `;`

ContinueStatement:

`continue` *Identifier*$_{opt}$ `;`

ReturnStatement:
`return` *Expression*$_{opt}$ `;`

ThrowStatement:
`throw` *Expression* `;`

SynchronizedStatement:
`synchronized (` *Expression* `)` *Block*

TryStatement:
`try` *Block Catches*
`try` *Block Catches*$_{opt}$ *Finally*

Catches:
CatchClause
Catches CatchClause

CatchClause:
`catch (` *FormalParameter* `)` *Block*

Finally:
`finally` *Block*

B.11 Expressions

Primary:
PrimaryNoNewArray
ArrayCreationExpression

PrimaryNoNewArray:
Literal
`this`
`(` *Expression* `)`
ClassInstanceCreationExpression
FieldAccess
MethodInvocation
ArrayAccess

ClassInstanceCreationExpression:
`new` *ClassType* `(` *ArgumentList*$_{opt}$ `)`

ArgumentList:

Expression
ArgumentList , *Expression*

ArrayCreationExpression:
`new` *PrimitiveType DimExprs* $Dims_{opt}$
`new` *ClassOrInterfaceType DimExprs* $Dims_{opt}$

DimExprs:
DimExpr
DimExprs DimExpr

DimExpr:
`[` *Expression* `]`

Dims:
`[ ]`
Dims `[ ]`

FieldAccess:
Primary . *Identifier*
`super` . *Identifier*

MethodInvocation:
Name `(` $ArgumentList_{opt}$ `)`
Primary . *Identifier* `(` $ArgumentList_{opt}$ `)`
`super` . *Identifier* `(` $ArgumentList_{opt}$ `)`

ArrayAccess:
Name `[` *Expression* `]`
PrimaryNoNewArray `[` *Expression* `]`

PostfixExpression:
Primary
Name
PostIncrementExpression
PostDecrementExpression

PostIncrementExpression:
PostfixExpression `++`

PostDecrementExpression:
PostfixExpression `--`

UnaryExpression:
PreIncrementExpression
PreDecrementExpression
`+` *UnaryExpression*

- *UnaryExpression*
UnaryExpressionNotPlusMinus

PreIncrementExpression:
++ *UnaryExpression*

PreDecrementExpression:
-- *UnaryExpression*

UnaryExpressionNotPlusMinus:
PostfixExpression
~ *UnaryExpression*
! *UnaryExpression*
CastExpression

CastExpression:
(*PrimitiveType* $Dims_{opt}$) *UnaryExpression*
(*Expression*) *UnaryExpressionNotPlusMinus*
(*Name Dims*) *UnaryExpressionNotPlusMinus*

MultiplicativeExpression:
UnaryExpression
MultiplicativeExpression * *UnaryExpression*
MultiplicativeExpression / *UnaryExpression*
MultiplicativeExpression % *UnaryExpression*

AdditiveExpression:
MultiplicativeExpression
AdditiveExpression + *MultiplicativeExpression*
AdditiveExpression - *MultiplicativeExpression*

ShiftExpression:
AdditiveExpression
ShiftExpression << *AdditiveExpression*
ShiftExpression >> *AdditiveExpression*
ShiftExpression >>> *AdditiveExpression*

RelationalExpression:
ShiftExpression
RelationalExpression < *ShiftExpression*
RelationalExpression > *ShiftExpression*
RelationalExpression <= *ShiftExpression*
RelationalExpression >= *ShiftExpression*
RelationalExpression `instanceof` *ReferenceType*

EqualityExpression:
RelationalExpression

EqualityExpression `==` *RelationalExpression*
EqualityExpression `!=` *RelationalExpression*

AndExpression:
EqualityExpression
AndExpression `&` *EqualityExpression*

ExclusiveOrExpression:
AndExpression
ExclusiveOrExpression ∧ *AndExpression*

InclusiveOrExpression:
ExclusiveOrExpression
InclusiveOrExpression | *ExclusiveOrExpression*

ConditionalAndExpression:
InclusiveOrExpression
ConditionalAndExpression `&&` *InclusiveOrExpression*

ConditionalOrExpression:
ConditionalAndExpression
ConditionalOrExpression || *ConditionalAndExpression*

ConditionalExpression:
ConditionalOrExpression
ConditionalOrExpression `?` *Expression* `:` *ConditionalExpression*

AssignmentExpression:
ConditionalExpression
Assignment

Assignment:
LeftHandSide AssignmentOperator AssignmentExpression

LeftHandSide:
Name
FieldAccess
ArrayAccess

AssignmentOperator: one of
`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `>>>=` `&=` ∧`=` |`=`

Expression:
AssignmentExpression

ConstantExpression:
Expression

References

[AB] Karine Arnout and Éric Bezault. How to get a Singleton in Eiffel. *Submitted for publication*. http://se.inf.ethz.ch/people/arnout/arnout_bezault_singleton.pdf.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BCE] Byte code engineering library (BCEL). http://jakarta.apache.org/bcel/.

[Bez] Éric Bezault. Gobo Eiffel project. http://www.gobosoft.com.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996. http://java.sun.com/docs/books/jls/.

[GM96] James Gosling and Henry McGilton. The java language environment, a white paper, May 1996. http://java.sun.com/docs/white/langenv/.

[Jav] Java 2 platform. http://java.sun.com.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999. http://java.sun.com/docs/books/vmspec/.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. 2nd edition, Prentice Hall, 1997.

[Sun] Sun microsystems. http://www.sun.com.